# Applying Neural Networks on the Number Partitioning Problem

Nicolas Cavrel

## Introduction

The problem of telling whether or not a finite set of integers can be partitioned into two subsets of equal sum is defined as the Number Partitioning Problem (NPP). Even thought there are very powerful heuristics able to solve a large number of instances, no algorithm can solve them all in polynomial time. Thus the problem is classified as an NP-hard problem.

The goal here is to have a look on how a Neural Network (NN) is able to perform on this problem, which gives us an example of application of Neural Networks on NP-hard problems.

## I. About Neural Networks

A Neural Network is a mathematical entity composed of a given number of Neurons linked together. Neurons can be connected to each others in all sorts of way (a neuron can even be connected with itself). The architecture of the connections is defining the kind of NN you're using. Based on this architecture we can define layers structuring the Network. We'll consider here $feedforward$ Neural Networks as the one schematized on Figure 1 :

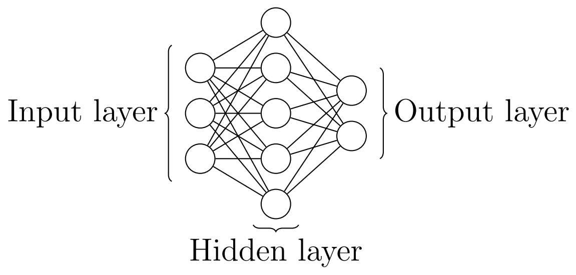


Figure 1.: A simple schematized feedforward NN

The layers of the $feedforward$ network are ordered and each neuron of the previous layer is linked to each neuron of the next one.
The very first layer is called the *Input layer* and here are inputed the entries of the NN. The last layer is the *Output layer* corresponding to the output of the NN.
In between these two layers are stacked one or several *Hidden layers* (only one on the example of Figure 1), which are computation intermediaries.

As a layer is only connected to the next one, the values in a layer only depend on the values of the previous one. In fact it is defined this way :

$$L^{(i+1)} = f^{(i)}(W^{(i)}L^{(i)} - B^{(i)})$$

Let $n_i$ be the size of layer $i$:

- $L^{(i)}$ is the vector containing the *values* of layer $i$, $L^{(i)} \in \mathbb{R}^{n_i}$;
- $W^{(i)}$ is the *weights* matrix of layer $i$ : $W^{(i)} \in M_{n_{i+1},n_i}(\mathbb{R})$ and $\forall (p,q) \in \{1,...,n_{i+1}\} \times \{1,...,n_i\}, W^{(i)}_{p,q}$ is the weight of the connection between neuron $q$ of layer $i$ and neuron $p$ of layer $i+1$;
- $B^{(i)}$ is the vector containing the *biases* of layer $i$, $B^{(i)} \in \mathbb{R}^{n_{i+1}}$;
- $f^{(i)}$ if the activation function of the layer $i$ which is applied element-wise, $f^{(i)}$ should be differentiable everywhere if we want to optimize the NN through a gradient method. Here we will use the *ReLU*, *sigmoid* or $softmax$ activation functions.

In order to make a NN learn how to associate an output to a given input, we compare the computed output value *y_pred* and the expected one *y_true* through a $loss\ function$ denoted $C$. $C$ can be seen as a measure of the difference between *y_pred* and *y_true*; minimizing $C(y\_pred, y\_true)$

means tuning the NN in order to predict a result closer to the expected one. The minimization of $C(y\_pred, y\_true)$ is made w.r.t. the weights and biases of the layers with a gradient optimization method. The chosen gradient method is the ADAM [2] one.

There are many ways to use a Neural Network. For instance, some use it in order to select the best heuristic to apply to a given instance [3]. Others are trying to model a linear program within a Neural Network [1]. There are even NN using *reinforcement learning* in order to learn everything by themselves [6, 4]. In the following, we will see two other ways to use a NN: as a classifier and as a move selecter.

## II. About the Number Partitioning Problem

The NPP is a NP-hard problem as there is still no algorithm solving it in a polynomial time. It is defined as such:

- Instance: a finite set of integers $L$ s.t. $|L| < +\infty$ and $\forall l \in L,\ l \in \mathbb{N}$;
- Question: is it possible to find two disjoint subset of $L$ of equal sum: we want to find $L_1,\ L_2 \subset L$ such as:

$$L_1 \cap L_2 = \emptyset \ and \sum_{l_1 \in L_1} l_1 = \sum_{l_2 \in L_2} l_2 \ and \ L_1 \cup L_2 = L$$

In order to simplify the problem, we will consider here lists having at most 20 integers (*ie* $|L| \leq 20$), included in $\{1, ..., 300\}$.
Although a dynamic programing algorithm exists and gives an exact solution of each instances, this algorithm is very costly complexity-wise (actually pseudo-polynomial). Thus we have heuristics returning the right answer in many cases [7].

### II.1. Specific features of the NPP

As we want to partition an integer list into two subsets of equal sum, the first thing to notice is that a list cannot be splitted in two if its sum is odd.
A list of integer has an odd sum if there is an odd number of odd integers. When generating a random integer (through a uniform law) in $\{1, ..., 300\}$, the probability of it being odd is exactly 50%.
So one can show that the probability of having an odd sum list is $\frac{\lfloor \frac{n+1}{2} \rfloor}{n} \approx \frac{1}{2}$ with $n$ big enough ($\lfloor x \rfloor$ is the integer part of $x$).
The second big feature of the NPP is the number of elements in the list. Intuitively we can think that the more elements you have within a list and the more likely it is to be partitionable. For instance a 2 elements list can only be partitioned if the two elements are equal, which is a very rare case. For a 3 elements list, it is partitionable only if one element is the sum of the two others.
An estimation of the pourcentage of partitionable instances w.r.t. the number of element within the list can be seen on Figure 2.

These two features seem to be quite heavily related to the problem, and we'll analyze our results through them.

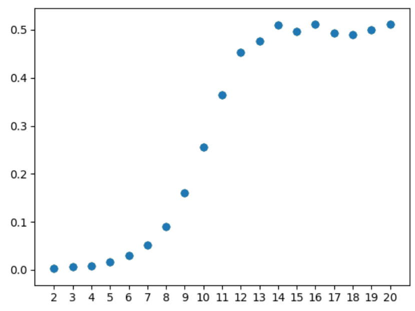


Figure 2.: Estimation of the pourcentage of partitionable instances w.r.t. the number of elements, estimated empirically on 5000 randomly generated instances.

## II.2. Instance generation

If we want to train a Neural Network on this problem, we need to generate a great number of solved instances. The first naive idea could be to generate randomly the instances and to solve them with the dynamic programing algorithm, but this method is very time consuming.
Thankfully, there is an algorithm building partitionable instances on a linear time (w.r.t $n$ the size of the list we want to build) :

```
Data: The size $n$ of the list we want to generate, $n > 1$
Result: A partitionable instance of the NPP
initialization;
$L = 0_{\Re^n}$;
$delta = 0$;
while $n > 1$ do
    $l = \text{randomInt } (\in \{1, ..., 300\})$;
    $L_n \leftarrow l$;
    if $delta \geq 0$ then
        $delta \leftarrow delta - l$
    else
        $delta \leftarrow delta + l$
    end
    $n \leftarrow n - 1$
end
$L_1 \leftarrow delta$;
return $L$;
```

**Algorithm 1:** Linear time algorithm generating *True* instances

# III. First NN approach : Classification Networks

## III.1. Classification Networks on the integer list

A first approach of the problem is to create a NN taking a list of integers as its input and which has only 2 outputs: *True* or *False*.

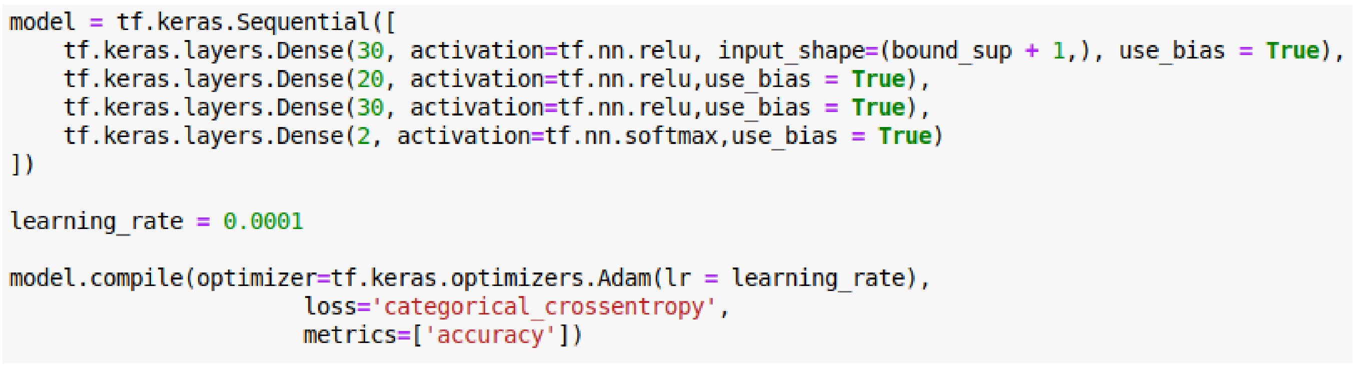
However, we will not give grossly the integer list to the network but a *formatted list*, which is the occurrence list of the initial list. Indeed, we want a list and its permutations to have the same output which is the case with an occurence list.

In order to train this NN, we will create a training data base containing a set of lists generated randomly each associated to a label (which can be either *True* or *False*) corresponding to its partitionability. We also need a test database which is similar to the training one but smaller ($\frac{1}{3}$ of the training one). The training process is as follows:

- An element is inputed into the network, and the output (the NN predicted answer) is computed;
- The difference between the expected and the predicted answer (called the loss value) is computed via the loss function (the bigger the loss value and the further away was the prediction from the expected answer);
- We compute the gradient of the loss function w.r.t. the parameters of the Network;
- A portion (the learning rate) of the gradient is subtracted to the parameters: it lowers the loss function and thus the new predicted answer will be closer to the expected one.

Once we've been through every element of the training set, we call it an epoch. Then the we compare the answers of the NN on the test set and save the proportion of correct answers in order to see the progression of the Network.

We will train here the NN on a 30 000 elements training base and on a 10 000 element test base. Here are the Network parameters (written with Keras).



```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(30, activation=tf.nn.relu, input_shape=(bound_sup + 1,), use_bias = True),
    tf.keras.layers.Dense(20, activation=tf.nn.relu,use_bias = True),
    tf.keras.layers.Dense(30, activation=tf.nn.relu,use_bias = True),
    tf.keras.layers.Dense(2, activation=tf.nn.softmax,use_bias = True)
])

learning_rate = 0.0001

model.compile(optimizer=tf.keras.optimizers.Adam(lr = learning_rate),
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
```

Figure 3.: Keras code defining the Neural Network and its optimization parameters

The results of the training are displayed below on Figure 4. We can see that if the NN is able to learn on the training set, it is not able to *apply* it on the test set: the accuracy on predicting the training set increases but the accuracy on the test set is decreasing.
In this case the network is fitting itself to the training data without learning to solve the main problem: this is overfitting.

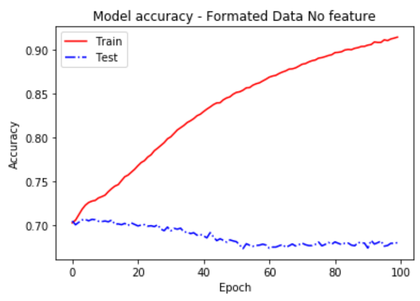


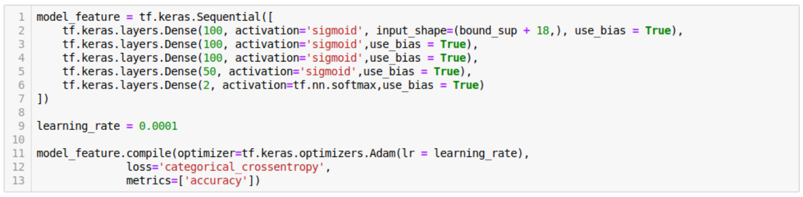
Figure 4.: Training of the NN on 100 epochs

## III.2. Classification Networks on the list with features

Giving only the integers list doesn't seem to be enough to teach things to the NN, that is why in this part we will try to give as an entry the integer list with a set of *features*.
These *features* are additional data about the instance. Here we will add:

- the *mean*, *median*, the first and third *quartile*, the *variance* of the list;
- the *biggest* and *smallest* element of the list;
- the *parity sign* of the list: a bit equal to 1 if the sum of the list is even and 0 otherwise.

The training process is composed of two steps. The first one is a 100 epochs training run with a learning rate set at $lr = 10^{-4}$, and the second one is also a 100 epochs session but with a learning rate smaller: $lr = 10^{-5}$. The shape and the training results are displayed on the figures 5 and 6.



```
model_feature = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation='sigmoid', input_shape=(bound_sup + 18,), use_bias = True),
    tf.keras.layers.Dense(100, activation='sigmoid',use_bias = True),
    tf.keras.layers.Dense(100, activation='sigmoid',use_bias = True),
    tf.keras.layers.Dense(50, activation='sigmoid',use_bias = True),
    tf.keras.layers.Dense(2, activation=tf.nn.softmax,use_bias = True)
])

learning_rate = 0.0001

model_feature.compile(optimizer=tf.keras.optimizers.Adam(lr = learning_rate),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

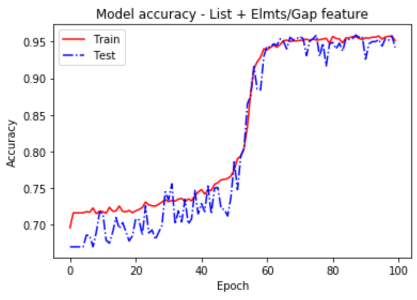
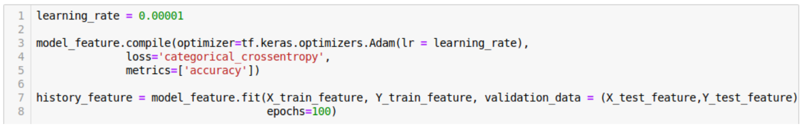


Figure 5.: Left: Shape and parameters of the NN. Right: First training run of the NN on 100 epochs



```
learning_rate = 0.00001

model_feature.compile(optimizer=tf.keras.optimizers.Adam(lr = learning_rate),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history_feature = model_feature.fit(X_train_feature, Y_train_feature, validation_data = (X_test_feature,Y_test_feature)
                                    epochs=100)
```

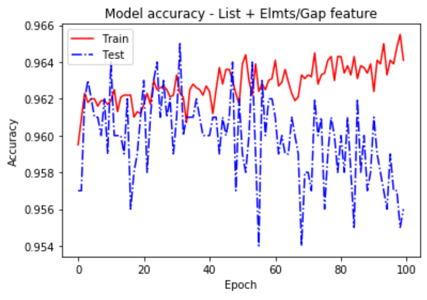
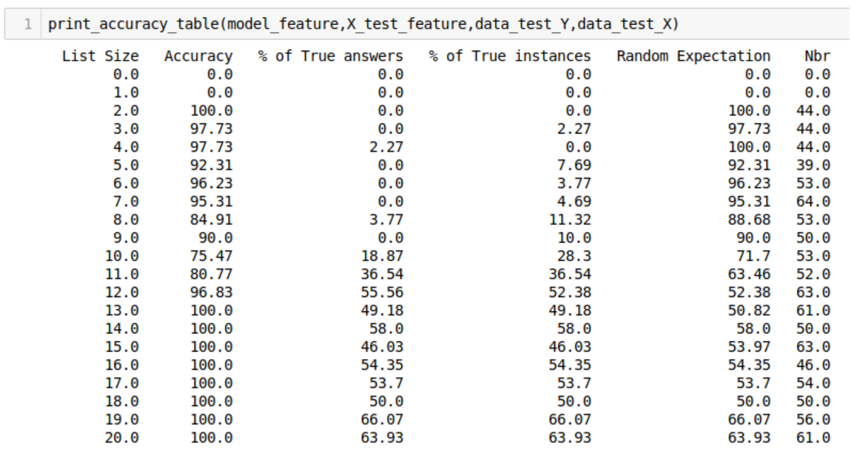


Figure 6.: Left: Lowering the learning rate. Right: Second training run of the NN on 100 epochs

As we can see on the graphs, the first training run is the one realizing the biggest improvement of the NN : its test accuracy goes from about 70% to about 95%. The second one is also improving the NN but in a smaller way (as the learning rate is 10 times smaller, this makes sense).
Even thought the second run is improving the Network accuracy, we can also see that some overfitting is appearing during the run. Maybe this shows the limits of the features we gave to the Network ?

In the end the Network is able to classify correctly about 95% of the instances, let's have a closer look on these results :
Overall, the accuracy is very high, except for the instances with a number of element in $7, ..., 12$ which correspond to the phase transition of the Figure 2 : on this figure, the instances with 6 or less elements are very unlikely to be *True* whereas the instances with 13 or more elements are very likely to be *True* (if their sum is even). The partitionability of the instances with a number of elements in between is harder to determine. These instances will be called the "hard" instances for the NN.



```
print_accuracy_table(model_feature,X_test_feature,data_test_Y,data_test_X)
```

```
  List Size   Accuracy  % of True answers  % of True instances  Random Expectation   Nbr
        0.0        0.0                0.0                  0.0                 0.0   0.0
        1.0        0.0                0.0                  0.0                 0.0   0.0
        2.0      100.0                0.0                  0.0               100.0  44.0
        3.0      97.73                0.0                 2.27               97.73  44.0
        4.0      97.73               2.27                  0.0               100.0  44.0
        5.0      92.31                0.0                 7.69               92.31  39.0
        6.0      96.23                0.0                 3.77               96.23  53.0
        7.0      95.31                0.0                 4.69               95.31  64.0
        8.0      84.91               3.77                11.32               88.68  53.0
        9.0       90.0                0.0                 10.0                90.0  50.0
       10.0      75.47              18.87                 28.3                71.7  53.0
       11.0      80.77              36.54                36.54               63.46  52.0
       12.0      96.83              55.56                52.38               52.38  63.0
       13.0      100.0              49.18                49.18               50.82  61.0
       14.0      100.0               58.0                 58.0                58.0  50.0
       15.0      100.0              46.03                46.03               53.97  63.0
       16.0      100.0              54.35                54.35               54.35  46.0
       17.0      100.0               53.7                 53.7                53.7  54.0
       18.0      100.0               50.0                 50.0                50.0  50.0
       19.0      100.0              66.07                66.07               66.07  56.0
       20.0      100.0              63.93                63.93               63.93  61.0
```

Figure 7.: Accuracy of the Network answers on the test set, w.r.t. the number of elements within the instances

### III.3. Conclusion on Classification Networks

The Classification Networks can be a very good indicator of the possible answer of the problem : it can have a very good accuracy on the general problem (95%) but is lacking in accuracy on the parts where the statistics doesn't help (especially on the hard instances). It also has a big drawback, it doesn't extract the solution of the problem. Thus we can refer to him as an indicator but we have to always doubt the veracity of its answer.

That is why we will now try to train instance solving NN instead of instance classifying NN.

## IV. Solving the NPP

### IV.1. About the format

In this part, the Network will be trying to solve instances of the NPP by extracting one of the objective subset. In order to do so, we have to define the way the Network will operate :

- We feed the Network with the first *state* of the game. A state is composed of the integers list plus some features we want to give to the Network. As a beginning we will consider that we give the Network the integers list with the objective value it has to reach (the integer part of the sum of the list divided by two), it looks like that:

$$\boxed{\underbrace{89 \quad 113 \quad 175 \quad 196 \quad 276 \quad 279}_{\text{Integers list}} \quad \underbrace{564}_{\text{Objective value}}}$$

- From this *state* the Network will return a *probability distribution* containing the probability of each element to be selected by the Network. This kind of Networks are called *policy networks*. Here is an example of the output of such a network :

$$\boxed{0.1 \quad 0.4 \quad 0.2 \quad 0.15 \quad 0.05 \quad 0.1}$$

  From there we select one of the elements according to the probabilities : We divide the $[0, 1]$ interval into 6 segments with sizes equal to the previous probabilities and then generate a random real number in this interval with a uniform law. The segment in which the real number is located will define the selected element.

- Then we apply the choice of the NN by swapping the selected element with a 0 and by updating the features value :

$$\left.\begin{array}{c}\boxed{89 \quad 113 \quad 175 \quad 196 \quad 276 \quad 279 \quad 564} \\ \\ \boxed{89 \quad 0 \quad 175 \quad 196 \quad 276 \quad 279 \quad 451}\end{array}\right\rangle \text{The second element is selected and set to 0}$$

Then we repeat the process on the new state until the objective value is either 0 or negative. If the objective value ends up at 0, then the problem is solved and the non-null element remaining in the state are the ones composing the subset we want to find. Else, we start the process again from the beginning, hoping the selected move will differ from the first time and find a better solution. The more tries we give to NN and the better the solution computed by the NN will be.
After we did all of our tries, we return a binary value equal to $True$ if the NN solved the instance and $False$ otherwise, coupled with a real number $s$ in $[0, 1]$ corresponding to the pourcentage of the objective we reached :
Let $m$ be the smallest objective value we reached and $M$ the initial objective value :

$$s = \frac{M - |m|}{M}$$

## IV.2. Training the Policy Network

As we saw earlier, this process requires the network to return a probability vector. This is done by setting the *Output layer* activation function as a $softmax$. This $\mathbb{R}^N \to \mathbb{R}^N$ function is defined as such :

$$\forall i \in \{1, ..., N\},\ \sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

One can check that the resulting vector is a probability distribution as every coordinate is positive and $\sum_{i=1}^{N} \sigma(z)_i = 1$.
Now we need to train our *policy network*. This training will be done by supervised training, which means that we will show to the Network the good elements to pick across a collection of examples. The training process is the following :

- Generate a partitionable instance with Algorithm 1, and extract the solution (this can be done with the dynamic programing algorithm [7] or by saving the integers that were either added or subtracted to the *delta* in Algorithm 1).

- Go through every element of the subset, let's say it's the element $i$ of the whole set then we will fit the output of the network with the vector

$$\boxed{0 \quad ... \quad 0 \quad \underbrace{1}_{i^{th}\text{ coordinate}} \quad 0 \quad ... \quad 0}$$

  This fitting is made through an unusual loss function $\varphi$ which is defined this way, for $y_{pred}$ the predicted vector and $y_{true}$ the expected one, and $<\ ,\ >$ the usual scalar product [5]:

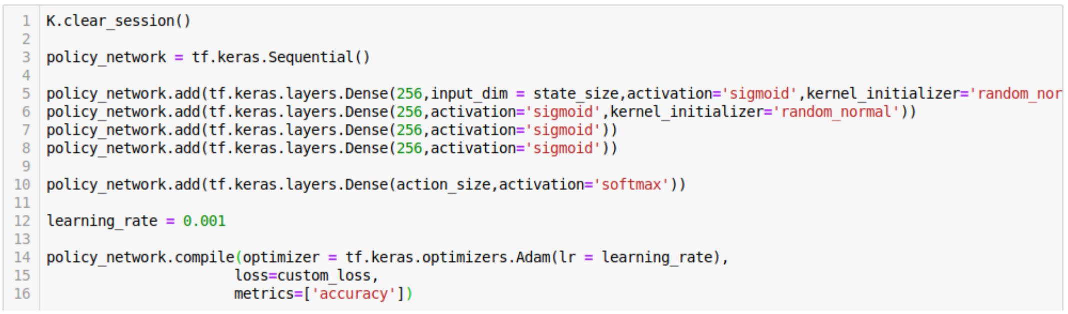
$$\varphi(y_{pred}, y_{true}) = -ln(< y_{pred}, y_{true} >)$$

Just to explain a bit this function, as every coordinate but one of $y_{true}$ is null, $< y_{pred}, y_{true} >$ is equal to the probability for the Network to pick the good element. As it's a real number in $[0, 1]$, $ln(< y_{pred}, y_{true} >)$ is negative and thus $\varphi(y_{pred}, y_{true})$ is positive.
Now the optimizer purpose is to minimize this loss value, which is the same as increasing the probability of picking the good element. This function should work as intended.
This loss function presents another interesting property : if the probability of picking the good element was very small, let's say close to 0, the loss value will be going toward $+\infty$ and thus the gradient will be large norm-wise. The change in the probability will be also great then. On the opposite, if the probability is close to 1, then the loss value will be close to 0 and the changes in the Network will be small.
This loss function depends of the previous probability of picking a move, if the Network was right, the changes will be close to null, but if the Network was completely wrong the changes will be large : this loss function allows the Network to learn new moves quickly.

This method requires a lot of examples to converge, so we will be doing several training sessions with different learning rates, each epochs being composed of the learning of all the elements to pick in an instance randomly generated through Algorithm 1.
Once every 1000 epochs the Network is tested and asked to solve 100 randomly generated instances in order to keep track of its improvements.

## IV.3. Training results

For this example we'll be training on list with 10 integers all contained in $\{1, ..., 300\}$ and the Neural networks parameters are in Figure 8 :



```
K.clear_session()

policy_network = tf.keras.Sequential()

policy_network.add(tf.keras.layers.Dense(256,input_dim = state_size,activation='sigmoid',kernel_initializer='random_nor
policy_network.add(tf.keras.layers.Dense(256,activation='sigmoid',kernel_initializer='random_normal'))
policy_network.add(tf.keras.layers.Dense(256,activation='sigmoid'))
policy_network.add(tf.keras.layers.Dense(256,activation='sigmoid'))

policy_network.add(tf.keras.layers.Dense(action_size,activation='softmax'))

learning_rate = 0.001

policy_network.compile(optimizer = tf.keras.optimizers.Adam(lr = learning_rate),
                       loss=custom_loss,
                       metrics=['accuracy'])
```

Figure 8.: The Policy Network parameters composed of 4 hidden layer each with sizes 256.

During the training sessions, we allow the NN to try 3 times solving the instance.
For the first training run, we did a 100000 epochs session, the results are displayed below :
After the first session, the Network is already really good at finding an approximated solution of the instance (the Network solution is at 99.4% of the objective value), however the proportion of instances solved to optimality is still not that high and oscillating around 22%.
The results of 6 of the 7 following training sessions are displayed on Figure 10. The learning rate is progressively decreased in order to allow the NN to learn smaller details of the data. Only the accuracy is displayed as the mean score isn't much evolving (it is starting very high : from 99.5% to 99.7% at the end of the training).
As the graphs show us, the NN seems to be improving through all the training process.

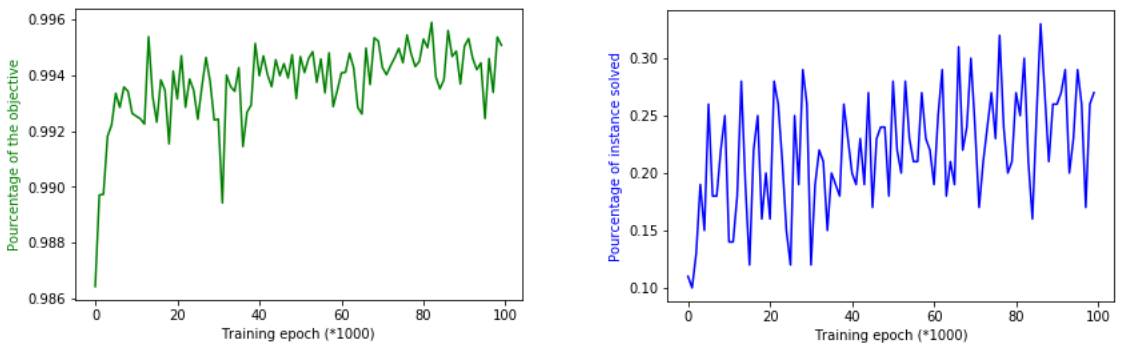
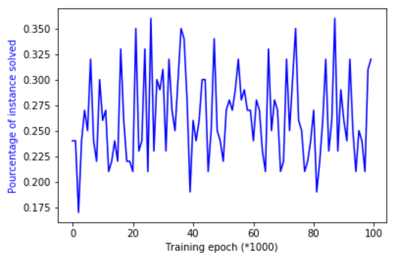
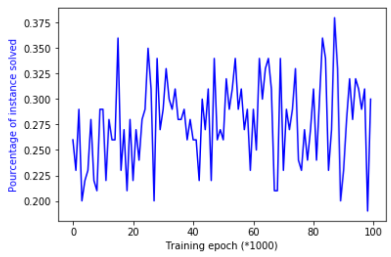


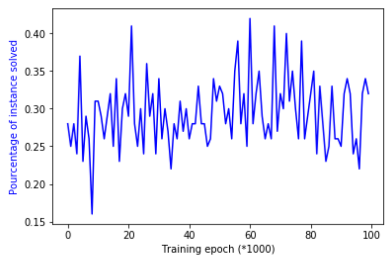
Figure 9.: First training session $lr = 10^{-3}$
Left: Evolution of the pourcentage $s$ of the objective value reached (see Section IV.1).
Right: Proportion of instances solved w.r.t. epochs



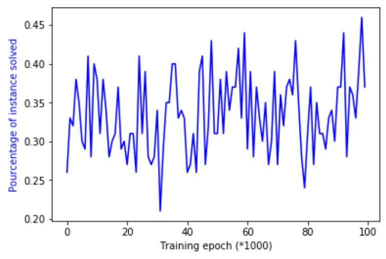
(a) 2nd training session, $lr = 10^{-3}$



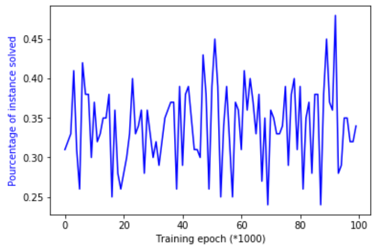
(b) 3rd training session, $lr = 10^{-4}$



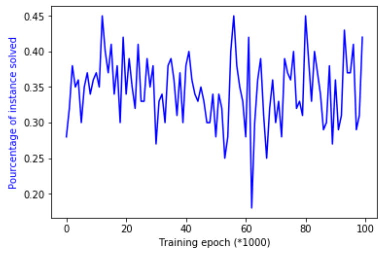
(c) 4th training session, $lr = 10^{-4}$



(d) 5th training session, $lr = 10^{-5}$



(e) 7th training session, $lr = 10^{-6}$



(f) 8th training session, $lr = 10^{-7}$

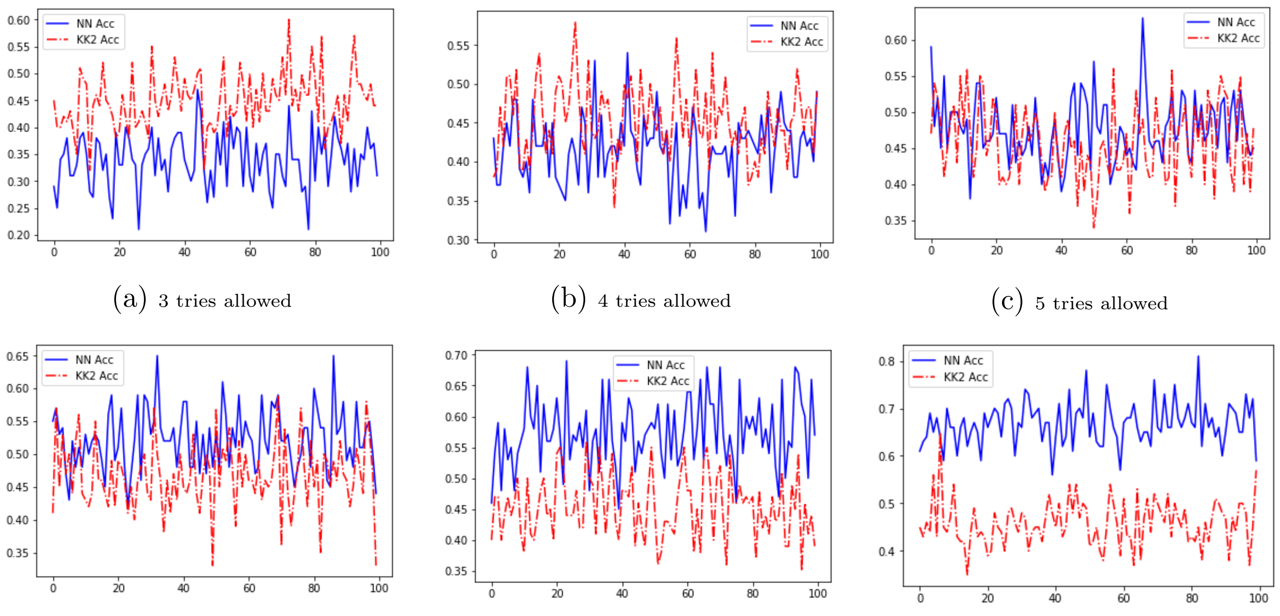
Figure 10.: The Accuracy of the NN through the training sessions.

## IV.4. Comparison with the Karmarkar-Karp heuristic

It is now time to compare our Neural Network performances with the known heuristics. In particular with the Karmarkar-Karp heuristic [7]. This heuristic is a good landmark for our Neural Network as both are biased toward the $True$ by attempting to extract a solution : if the heuristic or the NN answer $True$ we are sure of this answer (the extracted solution proves it), but if the answer is $False$ then we can't be sure of it. These two algorithms are doing pretty much the same job, it seems natural to compare them.

Figures 11 and 12 are displaying the Accuracy and the Score of the NN and the Karmarkar Karp heuristic over 10000 instances generated randomly (each point represents the mean over 100 instances).

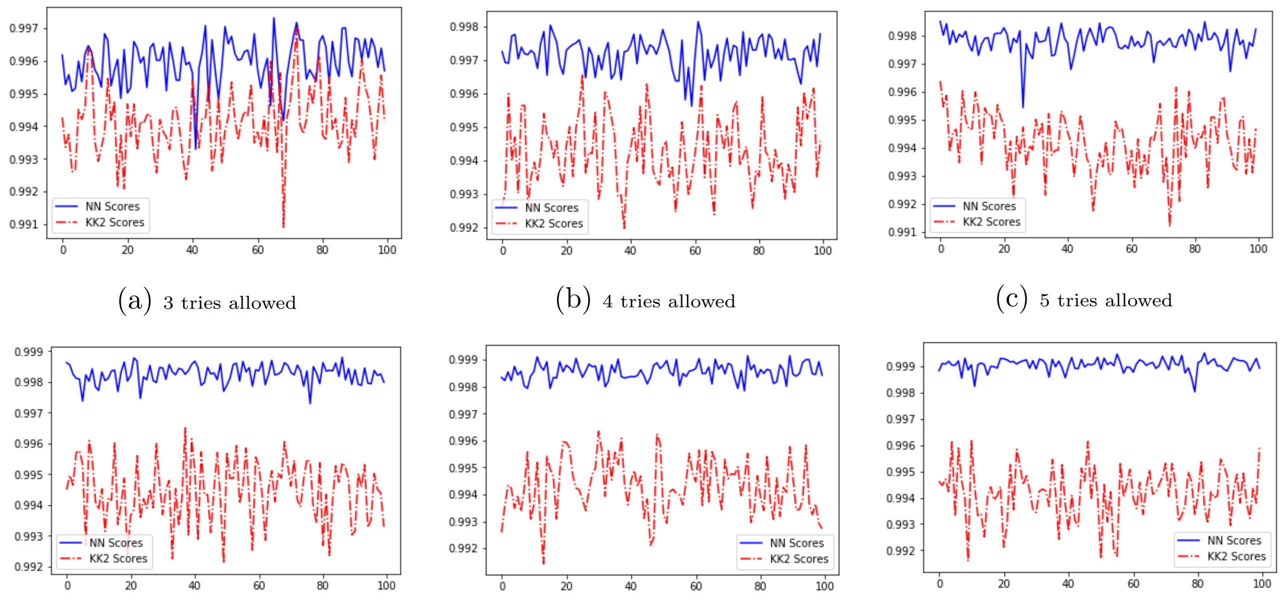
As we can see on Figures 11 and 12 the NN needs to have a big enough number of tries in order to reach the accuracy of the Karmarkar-Karp heuristic (here 5 tries for lists of size 10 and 1 try for lists of size 6). However, it is eventually beating it.



(a) 3 tries allowed (b) 4 tries allowed (c) 5 tries allowed

(d) 6 tries allowed (e) 7 tries allowed (f) 10 tries allowed

Figure 11.: The Accuracy of the NN and the Karmarkar-Karp heuristic (proportion of solved instances) with different numbers of tries allowed for the Network



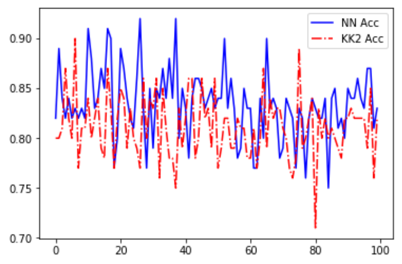
(a) 3 tries allowed (b) 4 tries allowed (c) 5 tries allowed

(d) 6 tries allowed (e) 7 tries allowed (f) 10 tries allowed

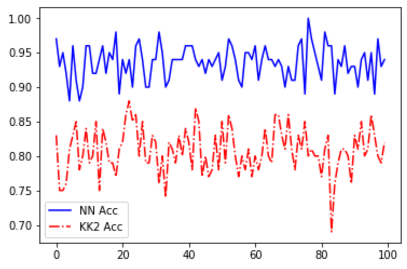
Figure 12.: The Score of the NN and the Karmarkar-Karp heuristic (in % of the objective value) with different numbers of tries allowed for the Network

As for the score, the NN is in average able to give better approximated solution than the heuristic pretty quickly, as only 3 tries are needed for lists of size 10.

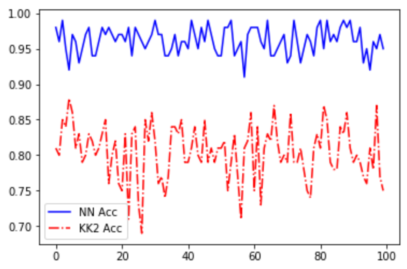
And on smaller lists, the Neural Networks are performing even better. Figure 13 shows the comparison between the Karmarkar-Karp heuristic and a similarly trained NN on lists of size 6.
For the instances of size 6, the NN has just better performances than the heuristic whatever the number of tries we grant it. Still, the more tries we grant it and the better the performances, and in this case, the Network is almost able to solve every instance (actually about 96%).



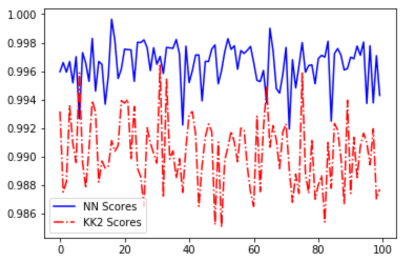
(a) Accuracy : 1 try allowed



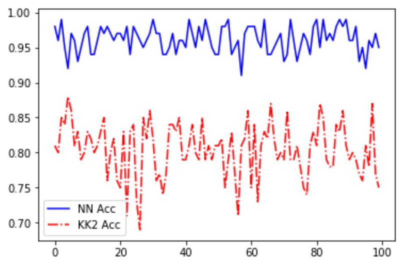
(b) Accuracy : 2 tries allowed



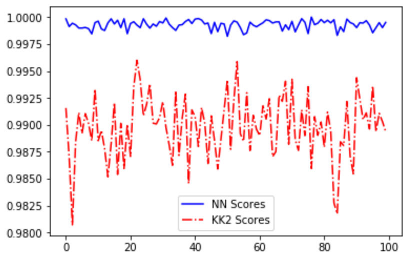
(c) Accuracy : 3 tries allowed



(d) Score : 1 try allowed



(e) Score : 2 tries allowed



(f) Score : 3 tries allowed

Figure 13.: Comparison between a NN and the Karmarkar-Karp heuristic for instances of size 6, with different values of allowed tries for the NN.

## Conclusion

As we saw, with the proper design and training, a Neural Network is able to outperform the current instance solving heuristic. However, it needs an increasing number of tries as the size of the problem increases, which slows down the process.
Where the Neural Networks do shine though, is when it is ask to find and approximated solution of the instance. In this case, it is able to find a solution very close to the real one pretty quickly.
One way to improve the current system would be to find better ways to exploit the probability distribution generated by the *policy network*. Right now, only one move is selected, but why wouldn't we try out the two most probable moves of the distribution ? Or we could try to detect whether or not the NN is sure of it's answer and pick the move(s) to try accordingly.

# Bibliography

[1] A. Agarwal, V. Jacob, and H. Pirkul. An improved augmented neural-network approach for scheduling problems. *INFORMS Journal on Computing*, 18:119–128, 02 2006.

[2] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[3] F. Mao, E. Blanco, M. Fu, R. Jain, A. Gupta, S. Mancel, R. Yuan, S. Guo, S. Kumar, and Y. Tian. Small boxes big data: A deep learning approach to optimize variable sized bin packing. *CoRR*, abs/1702.04415, 2017.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, Oct. 2017.

[7] T. Walshfipg. Analysis of heuristic for number partitioning. 2013.