Università degli studi di Padova

Computational Quantum Physics
Final assignment

Antonio Cusano
1203533

March 24, 2019

Abstract

The aim of this report is to develop machine learning algorithms for quantum physics models. In the first section we briefly recall some key concepts of machine learning. In the second one we show two conventional examples of tasks, i.e. text and images classification. Finally, we try to solve two classification problems in quantum domain, namely the distinction between phases in the Ising model and the separability testing of quantum states.

1 Machine learning theory

1.1 Basic concepts

Machine learning (ML) deals with systems that learn from the data how to perform specific tasks, without being explicitly programmed. These kind of algorithms can be used in a wide range of applications, such as email filtering and driving automation, hence there is a growing interest in this field.

In this report we will focus only on supervised learning approach, whose ultimate goal is to find an underlying input-output relation in the data, given some examples of input-output pairs (training data), in order to be able to predict an output from new input data. Here, without demanding completeness, we describe some very general concepts common to almost all ML problems.

Firstly, from a mathematical point of view we define an input space X and an output space Y, which inputs and outputs belong to. We will assume $X \subseteq \mathbb{R}^D$, whereas Y can be a subset of $\mathbb{R}^K$ as in regression case, or it can be a discrete set as in multiclass classification, $Y = \{1, 2, \ldots, K\}$. Significant is the case of binary classification where $Y = \{-1, 1\}$. The space $X \times Y$ is called data space.

We have to postulate the existence of a model for the data, so we assume that they are identically and independently sampled according to a fixed unknown distribution $p\left(x, y\right)$, with $x \in X$ and $y \in Y$. We underline that the goal of learning is not to find this whole distribution (which is ideal) but to estimate the best input-output relation.

In order to do that formally we have to fix a loss function $l : \; Y \times Y \to [0, \infty)$, $l\left(y, f(x)\right)$ is the cost of when predicting $f(x)$ instead of y. The expected loss is computed as

$$\mathcal{E}\left(f\right) = \mathbb{E}\left[l\left(y, f(x)\right)\right] = \int p\left(x, y\right) l\left(y, f(x)\right) \mathrm{d}x\mathrm{d}y$$

Then the best input-output relation is the target function $f^* : \; X \to Y$ that minimizes the expected error given l and p. Nevertheless it is impossible to find the target function f^* not having p.

Therefore we have to design a learning algorithm, that is a procedure which given, some training data $S = \{(x_n, y_n)\}_{n=1,\ldots,N}$, computes an estimator of the target function f_S. The most popular approach is the empirical risk minimization, which consists in considering as a proxy for the expected error the empirical error:

$$\hat{\mathcal{E}}\left(f\right) = \frac{1}{N} \sum_{n=1}^{n} l\left(y_n, f(x_n)\right)$$

In practice one has to fix a suitable hypothesis space H which f belong to and minimize $\hat{\mathcal{E}}$ over H, the latter process is called optimization.

To design a good algorithm one has to control two key concepts: fitting and stability. More precisely, a good estimator should fit data well but at the same time be stable, which means that its output should not change much due to slight changes in the input. In order to prevent overfitting (solutions highly dependent on the data) and ensure generalization often one has to apply the so called regularization techniques.

Most learning algorithms depend on one or more hyperparameters, that are parameters whose values are set before the learning process begins, and are not derived via training differently from other parameters. Hyperparameter optimization consists in finding the values which minimize the loss function on a given independent data set, namely the training set. Hence a common practice is to split the available dataset in a training set properly said, a validation set to tune hyperparameters, and a testing set on which evaluate the performance of the algorithm when it's completed.

1.2 Neural networks

We have just seen some vey basic concepts involved in ML theory from a purely theoretical point of view. Let's now discuss about classification problems from a more concrete point of view, starting from the linear classification approach.

Let's assume again that we have a training dataset $\{(x_n, y_n)\}_{n=1,\dots,N}$, where $x_n \in \mathbb{R}^D$ and $y_n \in \{1, 2, \dots, K\}$. Firstly, we define a linear score function $f : \mathbb{R}^D \to \mathbb{R}^K$ that maps the raw data to class scores:

$$f(x_n) = Wx_n + b$$

where W is a $K \times D$ matrix whose entries are called weights, and b a K dimensional vector called bias. Intuitively, one should look for the parameters values assign which for every input a big score to the true label and a small one to each other. Then we have to introduce a loss function[1] to quantify the agreement of the predicted class scores with real labels, and try to minimize it.

A popular choice is to firstly apply the softmax classifier. Denoting with f_i the i-th component of the scores vector, and omitting the dependence with respect to x_i and the parameters:

$$q_i = \frac{e^{f_i}}{\sum_{j=1}^{K} e^{f_j}}$$

which converts scores difficult to interpret in probabilities of belonging to each class (i.e. normalized to 1). After that, we can use as loss function the cross-entropy:

$$l(y_n, f(x_n,)) = H(p, q) = -\sum_{i=1}^{K} p_i \log q_i = -\log q_{y_n} = -f_{y_n} + \log\left(\sum_i e^{f_i}\right)$$

where $p_i = \delta_{i,y_n}$ is the true distribution (i.e. certainty to belong to the y_n class).

The next step is the optimization process, that is the empirical risk minimization with respect to the parameters W, b. Since in general the number of parameters can be very high it is not convenient to compute its gradient numerically, but in our case it is not very difficult to compute it analytically. Denoting for simplicity $\mathbf{w}_i = (W_{i1}, \dots, W_{iD}, b_i)$ and $\mathbf{x}_n = (x_n, 1)$, it is easy to show:

$$\nabla_{\mathbf{w}_i} \hat{\mathcal{E}}(W, b) = \sum_{n=1}^{N} (q_i - \delta_{i,y_n}) \mathbf{x}_n$$

In practice, initializing randomly the parameters, one can apply the gradient descent method, that is evaluate empirical risk gradient and update the parameters repeatedly in the opposite direction up to convergence[2]:

$$\mathbf{w}_i \to \mathbf{w}_i - \gamma \nabla_{\mathbf{w}_i} \hat{\mathcal{E}}(W, b)$$

[1] To be rigorous, in the previous paragraph we gave a slightly different definition of loss function, but its meaning is unchanged. Sometimes the term surrogate loss function is used.

[2] It can be shown that this algorithm converge. In more general non convex problems finding a global minimum can be much harder.

where γ is an hyperparameter named learning rate. An higher learning rate means faster convergence, but the optimization process could get stuck far from the minimum of the empirical risk.

Sometimes the training set can be very large, making it difficult to evaluate the gradient over all of it before every single parameters update. Hence, it is common to evaluate the gradient only on small batches of the training set. In the extreme case where the batch contains only one element the method is called stochastic gradient descent. In general, multiple passes over all the data are needed, each one is called epoch. The gradient descent method is only the simplest update method, but there are many more refined ones, for example Momentum, RMSprop, Adam. . . Finally, it can be helpful to anneal the learning rate over time (some of the previous methods already include an adaptive learning rate).

In the end, we need only the optimized parameters W, b in order to classify new inputs. In particular, we can forget the probably large training set. From a geometrical point of view, what we have just done (if possible) is to identify K hyperplanes that separate each class from the others in the $\mathbb{R}^D$ space.

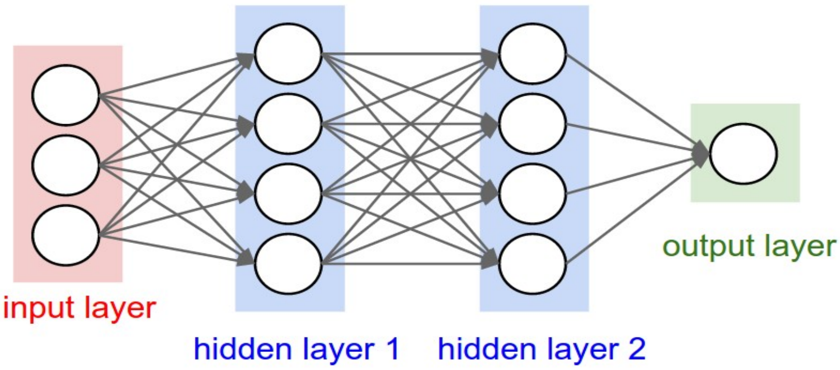


Figure 1: Example of neural network

How to deal with more complex problems? The basic idea of a neural network[3] (NN) is to compose simply parametrized representations:

$$f(x_n) = W\Phi(x_n) + b, \qquad \Phi = \phi_L \circ \cdots \circ \phi_2 \circ \phi_1$$

where, setting $d_0 = D$,

$$\phi_l = \sigma_l \circ W_l : \mathbb{R}^{d_{l-1}} \to \mathbb{R}^{d_l}, \quad W_l : \mathbb{R}^{d_{l-1}} \to \mathbb{R}^{d_l}, \quad \sigma_l : \mathbb{R} \to \mathbb{R}, \quad l = 1, \ldots, L$$

with W_l linear or affine transformation and σ_l non linear maps acting component-wise. Each intermediate representation corresponds to an hidden layer, and the dimensionality of a layer d_l corresponds to the number of units (or neuron) in it. The numbers of units and layers are other hyperparameters of the model.

A non linearity $\sigma(s)$ is called activation function. Commonly used examples are the sigmoid $\sigma(s) = (1+e^{-s})^{-1}$, the hyperbolic tangent $\sigma(s) = \tanh s$, and the rectified linear unit (ReLU) $\sigma(s) = \max(0, s)$.

Similarly, we have to minimize the empirical risk trough a gradient descent method, with respect to all the parameters W_l, W, b. Despite the big number of parameters, the

[3] We will implicitly consider only feedforward NN, wherein connections between the nodes do not form cycles.

gradient computation can be easily done using the chain rule for derivatives. More precisely the update consists in two steps: the forward pass, where functions outputs are computed from inputs, and the backward pass which starts at the end and recursively applies the chain rule to compute the gradients all the way to the inputs (the so-called back-propagation). It is fundamental to notice that generally the empirical risk minimization in a NN is not a convex problem, hence the convergence is not guaranteed.

We do not discuss here some important topics, such as parameters initialization, regularization schemes, hyperparameters tuning or data representation. We instead conclude describing some layers peculiar of image recognition problems. In fact, we just described layers where all the units are connected to each unit of the previous one, namely dense or fully connected layers (FC). On the other hand, the input space dimension can be too high for this kind of layers, causing overfitting or unmanageable number of parameters.

As example, the CIFAR-10 images used in the next section have dimensions $32 \times 32 \times 3$, corresponding to width, height and RGB color channels. A single unit in a fully connected layer would require 3072 weights (or 3073 parameters including the bias), still manageable. But using instead a full HD image ($1920 \times 1080 \times 3$) about 6 million parameters per unit would be needed, clearly untreatable.

The main problem is that FC layers do not take into account the spatial structure of the data, causing a waste of parameters. This issue is overcome by convolutional layers (Conv), which characterize a convolutional neural network (CNN). In this type of layer each unit is connected only to a small portion of the previous layer (namely its receptive field). The output of a unit is the dot product between the inputs in the receptive field and a set of weights of the same dimensions, which are the parameters to learn. Other units in the Conv layer can have a different receptive field, hence units are added up to cover completely the input size.

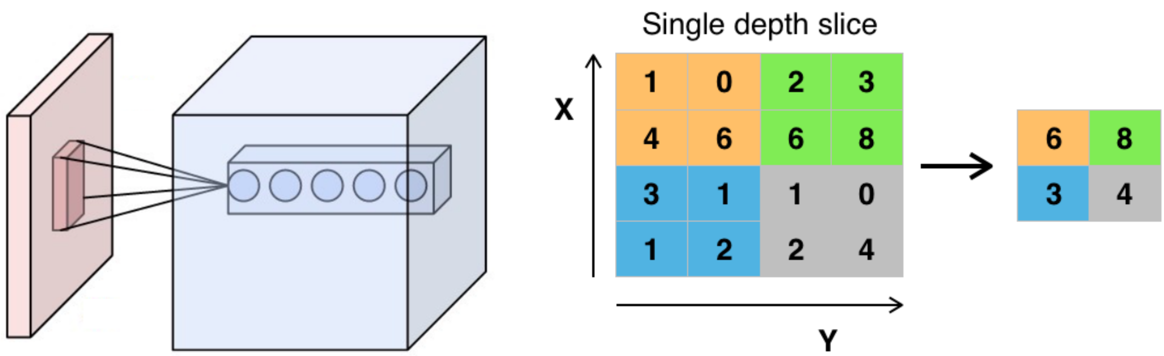


Figure 2: On the left a convolutional layer, on the right a max pooling layer (images from Wikipedia)

More precisely, thinking to have an image as input for clarity sake, e.g. a $32 \times 32 \times 3$ one as in CIFAR-10 dataset, the Conv layer is arranged as follows. Firstly we fix the size of the receptive field, e.g. $5 \times 5 \times 3$, observing that connectivity has to be local in width and height, but full along the input depth. Now we think to "look" (i.e. perform the dot product with the weights) a $5 \times 5 \times 3$ portion of the image, storing the output in a unit.

Then, we move the receptive field to another $5 \times 5 \times 3$ portion, denoting as stride the number of pixel we moved along, and store this output in an unit adjacent to the previous one. A reasonable stride choice is 1 or 2 pixel, meaning that the receptive fields of two adjacent units overlap. This procedure is iterated over the entire width and height of the input image. Eventually, adding zeros to the borders of the input image can be useful to obtain a desired shape of the output volume, a procedure known as zero padding.

We can replicate this procedure many times “looking” again at the image, adding new sets of weight associated to each input region, i.e. adding units that establish the depth of the output volume. The receptive field size, the stride, the zero-padding and the depth are all hyperparameters of the Conv layer (obviously there are some constrains on their combinations).

The parameters number can be drastically reduced using the parameters sharing. It consists in imposing to the units in a single slice of depth to share the same weights and bias. In this way, we can see the previously described procedure as a convolution of the units weights with the input volume, hence the sets of weights are commonly referred as filters or kernels. From a more creative viewpoint, we can imagine that each properly trained filter “responds ” when the input presents some specific type of feature (e.g. angle shapes, blobs of color . . .). This strengthen our hypothesis on parameters sharing.

Finally, most of CNNs include a pooling layer (often after a Conv one), which reduces the spatial size of the representation (and consequently the amount of parameters needed). In practice, working independently on each slice of depth, it partitions the input into a set of non-overlapping rectangles, then gives as output the maximum (or the average) computed on each of them. Advanced CNN architectures can be constituted of several stacked Conv, Pooling and FC layers.

1.3 TensorFlow and Keras libraries

TensorFlow is a free and open-source library for production and research in ML field. It was developed by the Google Brain team and released on November 2015. In this report we will use only its high level API Keras. The main advantages of this API are its user friendly interface, ideal for beginners, and its modularity and flexibility in building models.

Create a simple NN is very easy using the Sequential model. It consist in stacking the layers one by one with the add method. Most of the commonly used layers are already available, for example fully connected (Dense), convolutional (Conv2D, Conv1D, . . .), pooling (MaxPooling2D, AveragePooling2D. . .).

When a layer is added, one has to specify only its characteristic hyperparameters (e.g. the number of units in a FC). Only the first layer requires the input shape, whereas the following ones can compute it automatically. The activation functions can be specified when a layer is added or treated as independent layers (Activation). In this type of layer is included also the softmax classifier.

When the network is complete, through the Compile method the model is configured for the training. In particular, here one specifies the optimizer and the loss function. All the ones mentioned before, and much others, are available.

Finally, one has to train the model through the Fit method. All the processes of forward pass and backward propagation described in the previous section are automatically performed. An object containing the training history is returned. Once the model is trained we can evaluate its performances on a test dataset through the method Evaluate, which compares the predicted output with the real one, or we can use it to predict outputs from new data using Predict.

Keras provides also some data preprocessing functions (for text and images) and regularization schemes. Moreover, we mention that it is possible to customize building blocks, create new layers, loss functions and develop more complex models without much effort.

2 Conventional tasks

2.1 Text classification

Text classification is an interesting field where apply ML algorithms. Text is anywhere, hence it can be an interesting source of information but at the same time it can be very difficult to extract efficiently what we need. Text classification, i.e. assigning categories to text according to its content, is at the heart of spam filters, language identification and much more.

In this section, as warm up, we introduce an example of sentiment analysis, whose goal is to identify the type of opinion a text expresses. We will analyze the IMDb dataset, containing 50000 movie reviews from the Internet Movie Database, divided in half between training and test sets. The dataset contains an equal number of positive and negative reviews.

IMDb.py

```
from __future__ import absolute_import, division, print_function
import tensorflow as tf
from tensorflow import keras
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Activation, Embedding
from tensorflow.python.keras.layers import GlobalAveragePooling1D
from tensorflow.python.keras.preprocessing.sequence import pad_sequences
import numpy as np
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt
import os


# Downloads the IMDB dataset (bug in original file)
load_imdb = keras.datasets.imdb      # (see load_imdb.py)

# Keeps the top 10000 most frequent words in the training data
(train_data, train_labels), (test_data, test_labels)= \
    load_imdb.load_data(num_words=10000)
print('Data acquired! \n')
print("Training entries: {}, labels: {}".\
    format(len(train_data), len(train_labels)))

# A dictionary mapping words to an integer index
word_index = load_imdb.get_word_index()
word_index = {k:(v+3) for k,v in word_index.items()}
word_index["<PAD>"] = 0
word_index["<START>"] = 1
word_index["<UNK>"] = 2
word_index["<UNUSED>"] = 3
reverse_word_index = \
    dict([(value, key) for (key, value) in word_index.items()])
def decode_review(text):
    return ' '.join([reverse_word_index.get(i, '?') for i in text])

```

```
# Reads a review as example
user_input=input('Which review has to be shown? (write an int<25000)\n')
rev = int(user_input)
print(decode_review(train_data[rev]), train_labels[rev])

# Pads the arrays so they all have the same length
train_data = pad_sequences(train_data,
                            value=word_index["<PAD>"],
                            padding='post',
                            maxlen=256)
test_data = pad_sequences(test_data,
                           value=word_index["<PAD>"],
                           padding='post',
                           maxlen=256)

# Building the model
print('Building the model ...')

vocab_size = 10000
model = Sequential()
model.add(Embedding(input_dim=vocab_size, output_dim=16))
model.add(GlobalAveragePooling1D())
model.add(Dense(units=16))
model.add(Activation('relu'))
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['acc'])

print('Done! \n')
inp=raw_input('Print a summary of the model? (y/n)\t')
if(inp=='y'):
    model.summary()

# Training the model
raw_input("Press any key to start the training \n")
# Splits the data in training set and validation set
x_val = train_data[:10000]
partial_x_train = train_data[10000:]
y_val = train_labels[:10000]
partial_y_train = train_labels[10000:]

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=40,
                    batch_size=512,
                    validation_data=(x_val, y_val),
                    verbose=2)
history_dict = history.history
history_dict.keys()

# Evaluate the predictions on validation set
results = model.evaluate(test_data, test_labels)
print('On the test set the results are: \n')
print("Loss {}, accuracy {}".format(results[0], results[1]))
```

```
# Graphs of accuracy and loss over epochs
acc = np.array(history_dict['acc'])
val_acc = np.array(history_dict['val_acc'])
loss = np.array(history_dict['loss'])
val_loss = np.array(history_dict['val_loss'])

time = range(1, len(acc) + 1)
np.savetxt('history_imdb.txt',
            np.transpose((time, acc, val_acc, loss, val_loss)))

plt.figure(1)
plt.xlim(left=1, right=40)
plt.plot(time, loss, 'r', linewidth=1.5, label='Training')
plt.plot(time, val_loss, 'b', linewidth=1.5, label='Validation')
plt.title('Loss on IMDb')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.savefig('loss_imdb.pdf')

plt.figure(2)
plt.xlim(left=1, right=40)
plt.plot(time, acc*100.0, 'r', linewidth=1.5,label='Training')
plt.plot(time, val_acc*100.0, 'b', linewidth=1.5, label='Validation')
plt.title('Accuracy on IMDb', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Accuracy (%)', fontsize=14)
plt.legend()
plt.savefig('acc_imdb.pdf')
```

Let's briefly explain the code IMDb.py. First of all, the dataset is downloaded and imported by a script included in TensorFlow standard datasets, load_imdb.py[4]. In addition, it converts the words in integers through a dictionary containing the 10000 most recurrent words in the training data and saves the reviews as NumPy arrays.

The arrays are padded by a Keras function so they all have the same length. Then, this array (of shape number of reviews times maximum length) is sent to the Embedding layer, a type of layer peculiar of text classification algorithms. The idea behind an embedding layer is that we can represent words in a dense vector space, where the location and distance between them indicates how similar they are semantically. Hence it turns positive integers associated to the words into dense vectors of fixed size, which are learned during the model training. The dimension of this vector space can be seen as number of features.

Next, the global average GlobalAveragePooling1D, as the name suggest, averages over the word sequence dimension for each feature. Finally, we find two well known FC layers, and the network is closed by a sigmoid activation function. Then the model is trained as explained in the previous section, for 40 epochs. The execution was quite fast, taking about 30 minutes.

The performances on test and validation sets, monitored over the epochs, are shown in fig.3. On the other hand, on the test set our model scored an accuracy of 87.4%, proving its worth. We highlight that we have not used the validation set for hyperparameters tuning, but we kept it for further development.

[4] Actually the original name was imdb.py, but we changed it to avoid confusion with the main program. Fun fact: there was a typo in this code, "arrange" in place of "arange". We had to correct it to make the code run.

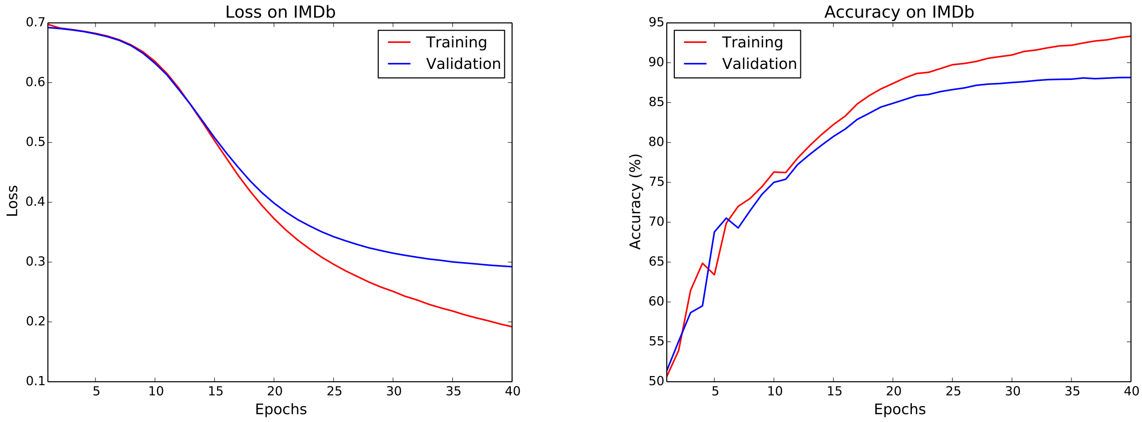
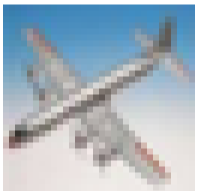
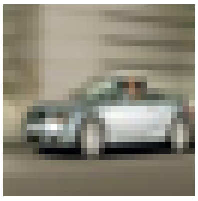


Figure 3: Performances on IMDb dataset

2.2 Images recognition

Computer vision is one of the most amazing fields where ML finds application. Practical implementation are uncountable and there are numerous contests on these topics, as the ImageNet Large Scale Visual Recognition Challenge. Noticeable was the 2012 edition where the advantages of the utilization of GPUs during training became clear. With 1.2 million 224×224 training color images in 1000 different categories, the winner network AlexNet achieved a classification error of 15.3%. Despite these impressive numbers, except all the technical details, this CNN is conceptually not very different from the ones we described in the previous section: multiple FC, Conv and MaxPooling layers stacked together.

In this section of the report we are going to analyze a much simpler images dataset. The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes equally populated. There are 50000 training images and 10000 test images, collected by the designers of AlexNet. The classes are completely mutually exclusive (see fig.4).


(a) Airplane

(b) Automobile

(c) Bird

(d) Cat

(e) Deer

(f) Dog

(g) Frog

(h) Horse

(i) Ship

(j) Truck

Figure 4: A picture per class in the CIFAR-10 dataset

We wrote the code CIFAR10.py as example of images recognition task. It can be thought as divided in four sections. In the first one the dataset is downloaded and imported in NumPy arrays by means of the script load_cifar10.py included[5] in Keras standard datasets (it can be found in the Appendix). The data are slightly preprocessed, that is the RGB pixel intensities are rescaled and centered in the range $[-0.5, 0.5]$, since it is a good habit to have small numbers centered on zero as inputs, and the labels encoding is converted from ordinal to one-hot (a sort of "binarization" of the categories).

In the second section the model is built as discussed in detail in the theory section. We only clarify that the padding option "same" results in padding the input such that the output has the same length as the original input, the Flatten layer just reshapes the tensor, and the Dropout layer is needed for the regularization. More precisely dropout consists in randomly ignoring (i.e. set to 0) a fraction of input units during each update of training, which reduces the risk of overfitting.

CIFAR10.py

```python
from __future__ import absolute_import, division, print_function
import numpy as np
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Dropout, Activation
from tensorflow.python.keras.layers import Conv2D, MaxPooling2D, Flatten
import os


num_classes = 10
save_dir = os.path.join(os.getcwd(), 'saved_results')
model_name = 'cifar10_trained_model.h5'
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)

# Loads the CIFAR-10 images dataset
load_cifar10 = keras.datasets.cifar10 # (see load_cifar10.py)
(train_imag, train_lab), (test_imag, test_lab)=load_cifar10.load_data()

print('Data acquired! \n')
print("Training images: {}, labels: {}".\
    format(train_imag.shape[0], train_lab.shape[0]))

# Convert class vectors to one-hot encoding
train_lab = keras.utils.to_categorical(train_lab, num_classes)
test_lab = keras.utils.to_categorical(test_lab, num_classes)

# Rescale the RGB values in the range [-0.5:0.5]
train_imag = train_imag.astype('float32')
test_imag = test_imag.astype('float32')
train_imag = train_imag/255.0-0.5
test_imag = test_imag /255.0-0.5

# Model of CNN
model = Sequential()
model.add(Conv2D(filters=32,
```

[5]Again, the name of the script was load_cifar10.py, but we renamed it to avoid confusion with the main program.

```
                  kernel_size=(3, 3),
                  padding='same',
                  input_shape=train_imag.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(rate=0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(units=512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))

opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

inp=raw_input('Print a summary of the model? (y/n)\t')
if(inp=='y'):
    model.summary()

# Training of the model
raw_input("Press any key to start the training \n")
history = model.fit(
        train_imag,
        train_lab,
        epochs=30,
        batch_size=32,
        validation_data=(test_imag, test_lab),
        verbose=1)

# Save the model and the weights
inp=raw_input('Save the model? (y/n)\n')
if(inp=='y'):
    model_path = os.path.join(save_dir, model_name)
    model.save(model_path)
    print('Saved trained model at %s ' % model_path)

history_dict = history.history
history_dict.keys()
acc = np.array(history_dict['acc'])
val_acc = np.array(history_dict['val_acc'])
loss = np.array(history_dict['loss'])
val_loss = np.array(history_dict['val_loss'])

time = range(1, len(acc) + 1)
history_path=os.path.join(save_dir, 'history_cifar10.txt')
```

```
np.savetxt(history_path,
           np.transpose((time, acc, val_acc, loss, val_loss)))

# Print and plot the results
print('Final validation accuracy: {}, loss: {}'. \
        format(val_acc[-1], val_loss[-1]))

plt.figure(1)
plt.xlim(left=1, right=30)
plt.plot(time, loss, 'r', linewidth=1.5, label='Training')
plt.plot(time, val_loss, 'b', linewidth=1.5, label='Validation')
plt.title('Loss on CIFAR-10',  fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.legend()
plt.savefig(os.path.join(save_dir, 'loss_cifar10.pdf'))

plt.figure(2)
plt.xlim(left=1, right=30)
plt.plot(time, acc*100.0, 'r', linewidth=1.5,label='Training')
plt.plot(time, val_acc*100.0, 'b',  linewidth=1.5, label='Validation')
plt.title('Accuracy on CIFAR-10', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Accuracy (%)', fontsize=14)
plt.legend()
plt.savefig(os.path.join(save_dir, 'acc_cifar10.pdf'))
```

The third section concern the training, where we set a batch size of 32 images and a total of 30 epochs, which took more than 12 hours to be completed. Actually, since we did not perform any hyperparameter tuning, there is no substantial difference between test set and validation set. Finally, in the last section the results are plotted and saved.

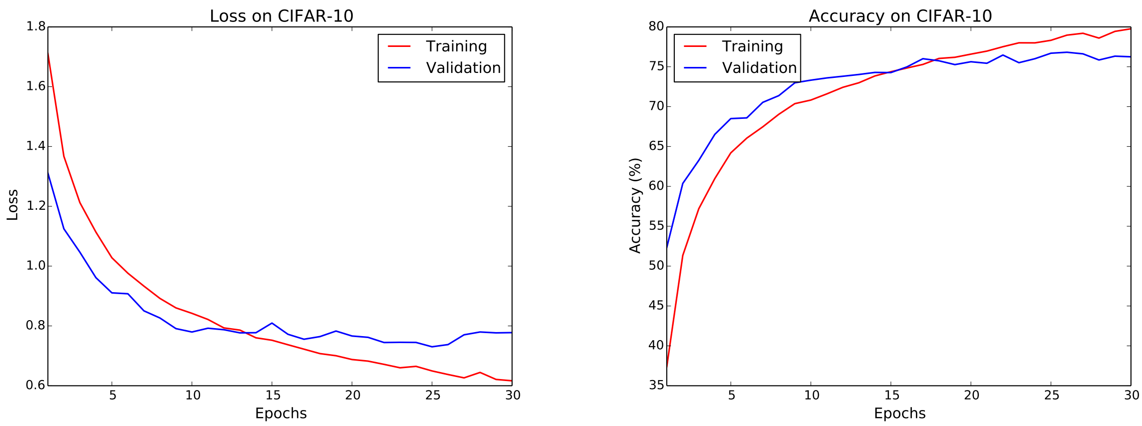


Figure 5: Performances on CIFAR-10 dataset

As we can see in fig.5 we reached an accuracy of 76.3% on unseen data. Not impressive, but still a good result for such a simple code. We notice that after approximately 10 epochs the accuracy on validation set stops increasing, whereas the results on training set continue to getting better. This is a clear symptom of overfitting. It would be interesting to find out if it is possible to reach the 80% of accuracy by hyperparameters optimization.

3 Quantum domain tasks

3.1 Ising model phases

We arrived to the most original part of this report. Now we will try to apply the ML algorithms we learned in the previous sections to the quantum domain.

Let's consider the one-dimensional transverse-field Ising model. Given N sites of spin $\frac{1}{2}$ in a one dimensional lattice, the hamiltonian which describes the system is

$$\hat{H} = \sum_{i=1}^{N} \sigma_z^i + \lambda \sum_{i=1}^{N-1} \sigma_x^i \sigma_x^{i+1}$$

where σ_x, σ_z are the Pauli matrices and λ is the interaction strength. Hence $\hat{H}$ is represented by a $2^N \times 2^N$ complex matrix.

We exploited the codes we wrote for Ex9 (see Appendices) to study the ground state energy of a system of size $N = 10$. More in detail, we sampled the ground state energy and the corresponding eigenvector (i.e. the ground state wavefunction) with $\lambda \in [-3, 3]$, collecting 6000 samples uniformly distributed.

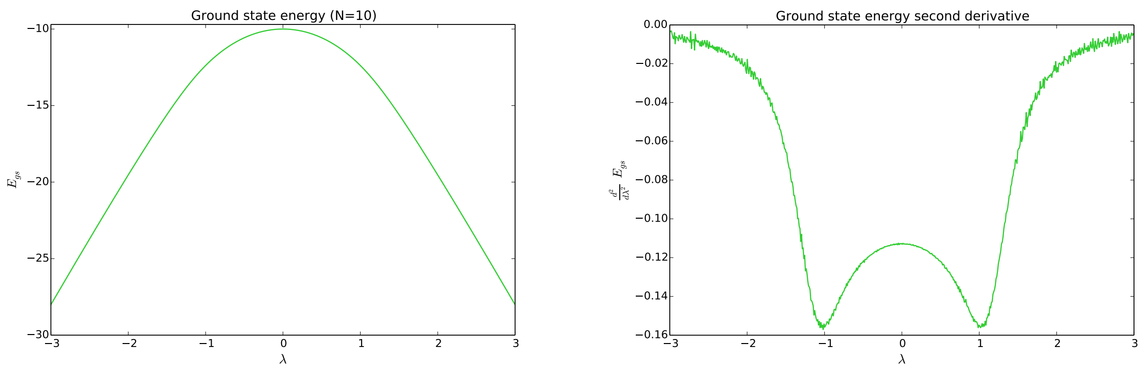


Figure 6: Exact solution of the 1D Ising model for $N = 10$

As we can see from fig.6 we expect a critical point in $|\lambda_C| = 1$ in the thermodynamic limit ($N \to \infty$). We can distinguish two phases: one for $|\lambda| < 1$ where the energy is quadratic in λ, and one for $|\lambda| > 1$ where the energy is linear in λ. We will refer to the first as disordered phase and to the second as ordered phase.

Our goal was to create a network capable of distinguish if a given wavefunction represents an ordered or disordered phase. As training dataset we used the vectors of a system with size $N = 10$ generated by the codes mentioned above, labelling each of them with 0 or 1 according to the corresponding value of λ.

ML_ising.py

```
from __future__ import absolute_import, division, print_function
import tensorflow as tf
from tensorflow import keras
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Dropout, Activation
from tensorflow.python.keras.layers import Conv1D, MaxPooling1D, Flatten
import numpy as np
import os
import matplotlib
```

```python
matplotlib.use('agg')
import matplotlib.pyplot as plt


# Monitores performances per batch (training set only)
class LossHistory(tf.keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.loss = []
        self.acc = []

    def on_batch_end(self, batch, logs={}):
        self.loss.append(logs.get('loss'))
        self.acc.append(logs.get('acc'))


save_dir = os.path.join(os.getcwd(), 'saved_results_ising')
model_name = 'ising_trained_model.h5'
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
model_path = os.path.join(save_dir, model_name)

seed=123
train_frac=0.7
val_frac=0.2

# Acquires the data
data_1=np.load('evec_dis_2000.npy')[:,1:]    # 0 < |lambda| < 1
data_2=np.load('evec_tra_2000.npy')[:,1:]    # 1 < |lambda| < 2
data_3=np.load('evec_ord_2000.npy')[:,1:]    # 2 < |lambda| < 3
data=np.concatenate((data_1,data_2,data_3), axis=0)

labels_1=np.ones(data_1.shape[0], dtype=int)
labels_2=np.zeros(data_2.shape[0], dtype=int)
labels_3=np.zeros(data_3.shape[0], dtype=int)
labels=np.concatenate((labels_1,labels_2,labels_3), axis=0)

Ntrain=int(train_frac*data.shape[0])
Nval=int(val_frac*data.shape[0])
Ntest=data.shape[0]-Ntrain-Nval

print('Data acquired! \n')

# Random shuffles the data
np.random.seed(seed)
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices] # Shuffle an array by row
labels = labels[indices]

# Splits the data in training, validation, test sets
data=data.reshape(data.shape[0],data.shape[1],1)
train_data=data[:Ntrain,:,:]
val_data=data[Ntrain:(Ntrain+Nval),:,:]
test_data=data[-Ntest:,:,:]
train_lab=labels[:Ntrain]
val_lab=labels[Ntrain:(Ntrain+Nval)]
test_lab=labels[-Ntest:]

```

```python
print("Total entries: {}\n Training:{}\n Validation: {}\n Test: {}". \
    format(data.shape[0], Ntrain, Nval, Ntest))

# Building the model
print('Building the model ...')
model = Sequential()

model.add(Flatten(input_shape=(1024,1)))
model.add(Dense(64))
model.add(Activation('relu'))

model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['acc'])

print('Done! \n')

inp=raw_input('Print a summary of the model? (y/n)\t')
if(inp=='y'):
    model.summary()

# Trains the model
raw_input("Press any key to start the training \n")

batch_history=LossHistory()
history = model.fit(train_data,
                    train_lab,
                    epochs=10,
                    batch_size=128,
                    validation_data=(val_data, val_lab),
                    callbacks=[batch_history],
                    verbose=1)

# Saves the model and the results
model.save(model_path)

history_dict = history.history
history_dict.keys()
acc = np.array(history_dict['acc'])
val_acc = np.array(history_dict['val_acc'])
loss = np.array(history_dict['loss'])
val_loss = np.array(history_dict['val_loss'])

time_ep = range(1, len(acc) + 1)
time_ba = range(1, len(batch_history.acc) + 1)

history_path=os.path.join(save_dir, 'history_ising.txt')
historyb_path=os.path.join(save_dir, 'historyb_ising.txt')
np.savetxt(history_path, np.transpose((time_ep, acc, val_acc,
                                       loss, val_loss)))
np.savetxt(historyb_path, np.transpose((time_ba,
                                        batch_history.acc,
                                        batch_history.loss)))

# Prints the results
```

```
print('Final validation accuracy: {}, loss: {}'.format(
        val_acc[-1], val_loss[-1]))
test_loss, test_acc = model.evaluate(test_data, test_lab)
print('Test accuracy:', test_acc)

# Graphs of accuracy and loss over time (in epochs and batches)
plt.figure(1)
plt.xlim(left=1, right=10)
plt.plot(time_ep, loss, 'r', linewidth=1.5, label='Training')
plt.plot(time_ep, val_loss, 'b', linewidth=1.5, label='Validation')
plt.title('Loss on Ising', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.legend()
plt.savefig('loss_ising.pdf')

plt.figure(2)
plt.xlim(left=1, right=10)
plt.plot(time_ep, acc*100.0, 'r', linewidth=1.5, label='Training')
plt.plot(time_ep, val_acc*100.0, 'b', linewidth=1.5, label='Validation')
plt.title('Accuracy on Ising', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Accuracy (%)', fontsize=14)
plt.legend()
plt.savefig('acc_ising.pdf')

plt.figure(3)
plt.plot(time_ba, batch_history.loss, 'r',
            linewidth=1.5, label='Training')
plt.title('Loss per batch on Ising', fontsize=16)
plt.xlabel('Batch', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.legend()
plt.savefig('batch_loss_ising.pdf')

plt.figure(4)
plt.plot(time_ba, np.array(batch_history.acc)*100.0, 'r',
            linewidth=1.5, label='Training')
plt.title('Accuracy per batch on Ising', fontsize=16)
plt.xlabel('Batch', fontsize=14)
plt.ylabel('Accuracy (%)', fontsize=14)
plt.legend()
plt.savefig('batch_acc_ising.pdf')
```

The code ML_ising.py is inspired by the ones discussed before. Since this was a completely new task, we had no idea of which was the model to use, hence we started with a very simple model, containing just a FC layer with ReLU activation function and a final FC layer with sigmoid activation, since it was a binary classification problem.

Surprisingly, in spite of its simplicity, this architecture provided quickly excellent results. As we can see in fig.7 our model reached an accuracy of almost 100% just after 5 epochs on both validation and training sets. Precisely, on the test set our model scored an accuracy of 99.7%, with an execution time shorter than 30s. In order to understand better the evolution of the model we added a function to monitor the performances after each batch. Unfortunately the fit method does not evaluate the results on validation set after each batch, but only at the end of the epochs. Hence in fig.8 only the training set is monitored.

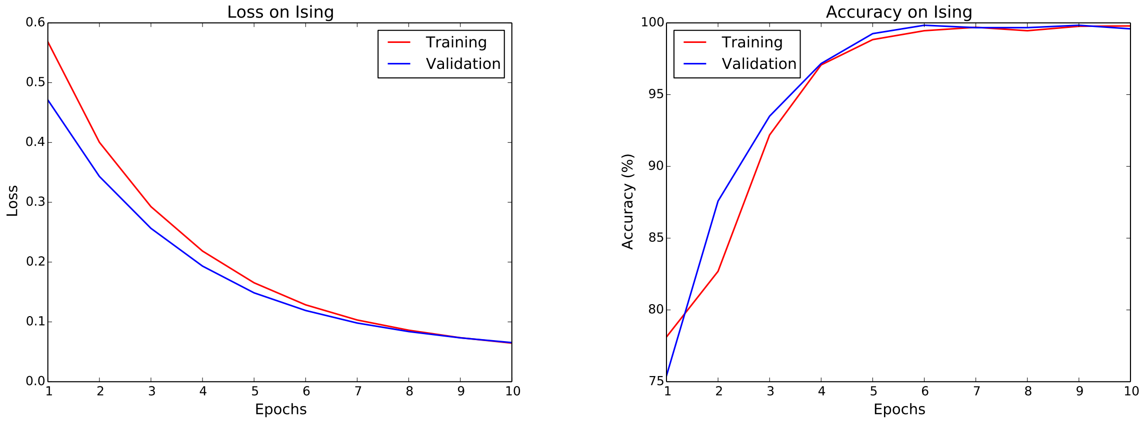


Figure 7: Performances on classification of the Ising model phases

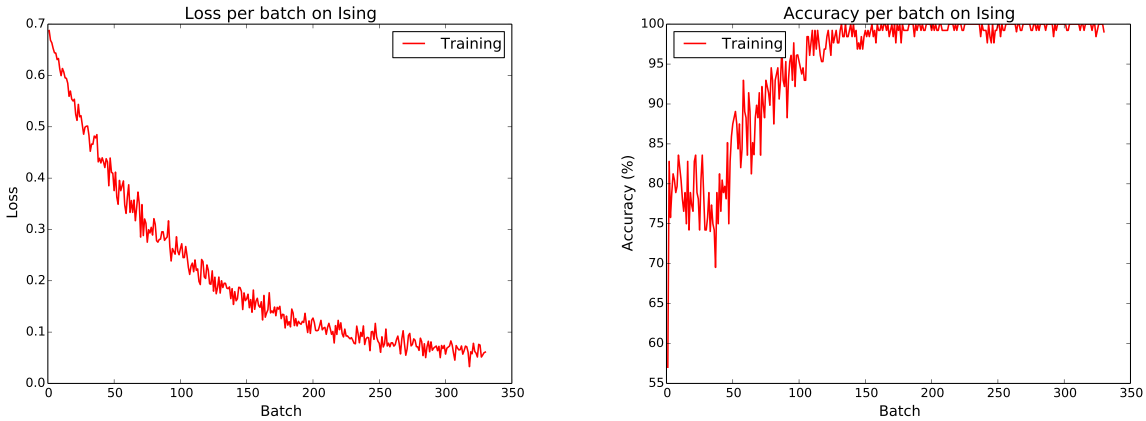


Figure 8: Performances per batch on the Ising model

3.2 Separability of quantum states

In the end we try to complete an even more interesting task. Given two Hilbert spaces $\mathcal{H}_1$, H_2, a pure state $|\psi\rangle \in H_1 \otimes H_2$ is said to be separable if it can be written as $|\psi_1\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$, with $|\psi_1\rangle$, $|\psi_2\rangle$ pure states of the respective subsystems. Otherwise, ψ is called entangled, and it is not possible to assign states to its subsystems. This definition can be generalized easily in the case of a quantum system composed of N subsystems. Separability testing in a general case is an NP-hard problem.

Our final goal is to create a network able to classify a given wavefunction as describing a separable or an entangled state. We used the script wfgen.py included in Appendices to generate a dataset. In particular we created 4000 examples of separable wavefunctions, performing the tensor product between 10 2-dimensional (complex) wavefunctions, each one randomly initialized according to a standard normal distribution and normalized. The 4000 examples of entangled wavefunctions are easily generated randomly initializing 1024-dimensional vectors, again according to standard normal and normalized. In fact, almost surely this procedure should give an entangled wavefunction.

ML_separability.py

```
from __future__ import absolute_import, division, print_function
import tensorflow as tf
from tensorflow import keras
```

```
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Dropout, Activation
from tensorflow.python.keras.layers import Conv1D, MaxPooling1D, Flatten
import numpy as np
import os
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt


save_dir = os.path.join(os.getcwd(), 'saved_results_wave')
model_name = 'separable_trained_model.h5'
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
model_path = os.path.join(save_dir, model_name)

seed=123
train_frac=0.7
val_frac=0.2

# Acquires the data
data_1=np.load('sep.npy')
data_2=np.load('nsep.npy')
data=np.concatenate((data_1,data_2), axis=0)

labels_1=np.zeros(data_1.shape[0], dtype=int)
labels_2=np.ones(data_2.shape[0], dtype=int)
labels=np.concatenate((labels_1,labels_2), axis=0)

print('Data acquired! \n')

# Random shuffle the data
np.random.seed(seed)
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices] # Shuffle an array by row
labels = labels[indices]

# Split the data in training, validation, test
Ntrain=int(train_frac*data.shape[0])
Nval=int(val_frac*data.shape[0])
Ntest=data.shape[0]-Ntrain-Nval

train_data=data[:Ntrain,:,:]
val_data=data[Ntrain:(Ntrain+Nval),:,:]
test_data=data[-Ntest:,:,:]
train_lab=labels[:Ntrain]
val_lab=labels[Ntrain:(Ntrain+Nval)]
test_lab=labels[-Ntest:]

print("Total entries: {}\n Training:{}\n Validation: {}\n Test: {}". \
    format(data.shape[0], Ntrain, Nval, Ntest))
print(len(test_lab))

# Building the model
print('Building the model ...')
model = Sequential()
"""
```

```
# Model 1, discarded
model.add(Flatten(input_shape=(1024,2)))
model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dense(128))
model.add(Activation('relu'))
"""
# Model 2
model.add(Conv1D(32, 3, padding='same', input_shape=(1024,2)))
model.add(Activation('relu'))
model.add(Conv1D(32, 3))
model.add(Activation('relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.25))

model.add(Dense(1))
model.add(Activation('sigmoid'))

opt = tf.keras.optimizers.Adam(lr=0.0015, decay=6e-6)
model.compile(optimizer=opt,
              loss='binary_crossentropy',
              metrics=['acc'])

print('Done! \n')

inp=raw_input('Print a summary of the model? (y/n)\t')
if(inp=='y'):
    model.summary()

# Trains the model
raw_input("Press any key to start the training \n")

history = model.fit(train_data,
                    train_lab,
                    epochs=10,
                    batch_size=128,
                    validation_data=(val_data, val_lab),
                    verbose=1)

# Saves the model
history_dict = history.history
history_dict.keys()
acc = np.array(history_dict['acc'])
val_acc = np.array(history_dict['val_acc'])
loss = np.array(history_dict['loss'])
val_loss = np.array(history_dict['val_loss'])
time_ep = range(1, len(acc) + 1)
history_path=os.path.join(save_dir, 'history_wave.txt')
np.savetxt(history_path, np.transpose((time_ep, acc, val_acc,
                                       loss, val_loss)))

# Prints the results
print('Final validation accuracy: {}, loss: {}'.format(
```

```
        val_acc[-1], val_loss[-1]))
test_loss, test_acc = model.evaluate(test_data, test_lab)
print('Test accuracy:', test_acc)

# Graphs of accuracy and loss over time
plt.figure(1)
plt.xlim(left=1, right=10)
plt.plot(time_ep, loss, 'r', linewidth=1.5, label='Training')
plt.plot(time_ep, val_loss, 'b', linewidth=1.5, label='Validation')
plt.title('Loss on separability', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Loss', fontsize=14)
plt.legend()
plt.savefig('loss_wave.pdf')

plt.figure(2)
plt.xlim(left=1, right=10)
plt.plot(time_ep, acc*100.0, 'r', linewidth=1.5, label='Training')
plt.plot(time_ep, val_acc*100.0, 'b', linewidth=1.5, label='Validation')
plt.title('Accuracy on separability', fontsize=16)
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Accuracy (%)', fontsize=14)
plt.legend()
plt.savefig('acc_wave.pdf')
```

As first attempt we tried an architecture with only three FC layers. As we can see in fig.9 it is clear that it was completely inappropriate. The model only learns the patterns characteristic of the training set, but it is unable to generalize. The loss on the validation set is even increasing, and the 50% accuracy is not better than a completely random classification.

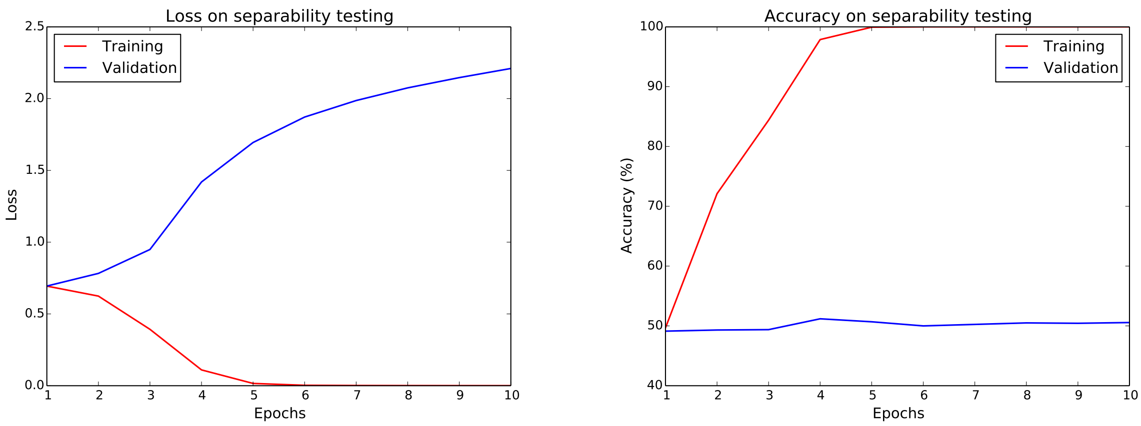


Figure 9: Performances on separability testing (discarded architecture)

Table 1: Architecture used on separability testing

Therefore, we decided to try an architecture similar to the one used on the CIFAR-10 dataset. This idea was inspired by an analogy between the 3 RGB channels and the wavefunction decomposition in real and imaginary part. Hence, we set up an architecture containing some (one-dimensional) Conv layers. It was astonishing to find out that this CNN was able to classify the test wavefunctions with an accuracy of 99.8%, after 10 minutes of training (about a minute per epoch).

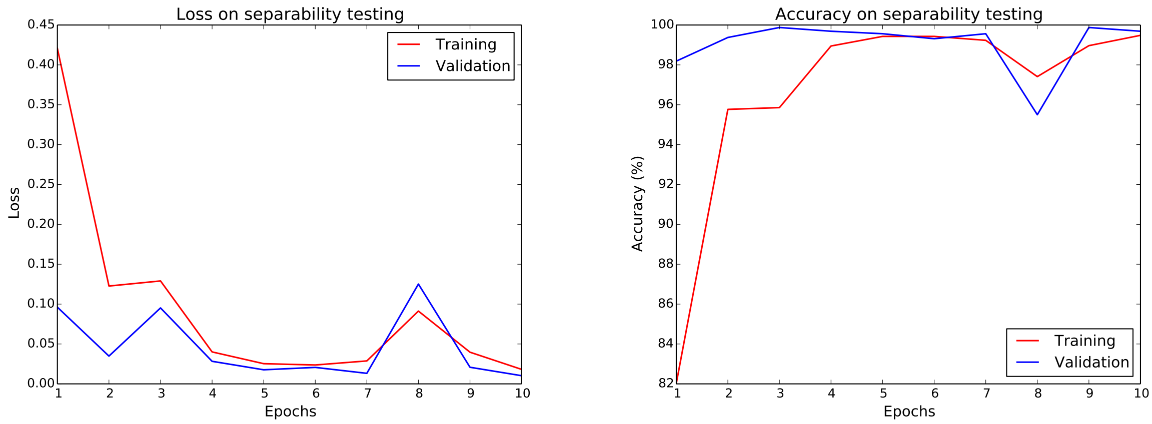


Figure 10: Performances on separability testing

Here we conclude our small journey in the world of ML. It was laborious, but we enjoyed it. We learned most of the basis of ML algorithms, and also how to implement some of them without much effort trough high level APIs, at least in the case of classification problems. As we anticipated, because of lack of time we could not treat, neither in theory nor in practice, some important topics, especially data representation, regularization schemes and hyperparameters tuning. Hence we hope to fill this gap in the next future, and enhance the skills we acquired.

4 References

Since ML is a relatively young and rapidly evolving field it was not easy to find a way in the literature. Our principal strategy was to search on the web from time to time. Most of the theoretical basis were apprehended in the following websites:

- Notes for the Stanford computer science class on CNN for visual recognition
  http://cs231n.github.io/convolutional-networks/
- Notes for machine learning crash course by L. Rosasco, hosted by Scuola Galileiana
  http://lcsl.mit.edu/courses/mlcc/mlcc2018/

For the practical implementation instead we followed mainly the tutorials on the websites of the libraries:

- TensorFlow
  https://www.tensorflow.org/tutorials/
- Keras
  https://keras.io/getting-started/sequential-model-guide/#examples

Appendices

load_imdb.py

```
# Copyright 2017 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================================
"""IMDB movie review sentiment classification dataset.
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import json

import numpy as np
from six.moves import zip  # pylint: disable=redefined-builtin

from tensorflow.python.keras._impl.keras.utils.data_utils import get_file


def load_data(path='imdb.npz',
              num_words=None,
              skip_top=0,
              maxlen=None,
              seed=113,
              start_char=1,
              oov_char=2,
              index_from=3):
  """Loads the IMDB dataset.

  Arguments:
      path: where to cache the data (relative to `~/.keras/dataset`).
      num_words: max number of words to include. Words are ranked
          by how often they occur (in the training set) and only
          the most frequent words are kept
      skip_top: skip the top N most frequently occurring words
          (which may not be informative).
      maxlen: sequences longer than this will be filtered out.
      seed: random seed for sample shuffling.
      start_char: The start of a sequence will be marked with this character.
          Set to 1 because 0 is usually the padding character.
```

```
        oov_char: words that were cut out because of the `num_words`
            or `skip_top` limit will be replaced with this character.
        index_from: index actual words with this index and higher.

    Returns:
        Tuple of Numpy arrays: `(x_train, y_train), (x_test, y_test)`.

    Raises:
        ValueError: in case `maxlen` is so low
            that no input sequence could be kept.

    Note that the 'out of vocabulary' character is only used for
    words that were present in the training set but are not included
    because they're not making the `num_words` cut here.
    Words that were not seen in the training set but are in the test set
    have simply been skipped.
    """
    path = get_file(
        path,
        origin='https://s3.amazonaws.com/text-datasets/imdb.npz',
        file_hash='599dadb1135973df5b59232a0e9a887c')
    f = np.load(path)
    x_train, labels_train = f['x_train'], f['y_train']
    x_test, labels_test = f['x_test'], f['y_test']
    f.close()

    np.random.seed(seed)
    indices = np.arange(len(x_train))
    np.random.shuffle(indices)
    x_train = x_train[indices]
    labels_train = labels_train[indices]

    indices = np.arange(len(x_test))
    np.random.shuffle(indices)
    x_test = x_test[indices]
    labels_test = labels_test[indices]

    xs = np.concatenate([x_train, x_test])
    labels = np.concatenate([labels_train, labels_test])

    if start_char is not None:
      xs = [[start_char] + [w + index_from for w in x] for x in xs]
    elif index_from:
      xs = [[w + index_from for w in x] for x in xs]

    if maxlen:
      new_xs = []
      new_labels = []
      for x, y in zip(xs, labels):
        if len(x) < maxlen:
          new_xs.append(x)
          new_labels.append(y)
      xs = new_xs
      labels = new_labels
      if not xs:
        raise ValueError('After filtering for sequences shorter than
   maxlen=' +
                         str(maxlen) + ', no sequence was kept. ')
```

```
                                 'Increase maxlen.')
    if not num_words:
        num_words = max([max(x) for x in xs])

    # by convention, use 2 as OOV word
    # reserve 'index_from' (=3 by default) characters:
    # 0 (padding), 1 (start), 2 (OOV)
    if oov_char is not None:
        xs = [[oov_char if (w >= num_words or w < skip_top) else w for w in
 x]
              for x in xs]
    else:
        new_xs = []
        for x in xs:
            nx = []
            for w in x:
                if skip_top <= w < num_words:
                    nx.append(w)
            new_xs.append(nx)
        xs = new_xs

    x_train = np.array(xs[:len(x_train)])
    y_train = np.array(labels[:len(x_train)])

    x_test = np.array(xs[len(x_train):])
    y_test = np.array(labels[len(x_train):])

    return (x_train, y_train), (x_test, y_test)


def get_word_index(path='imdb_word_index.json'):
    """Retrieves the dictionary mapping word indices back to words.

    Arguments:
        path: where to cache the data (relative to `~/.keras/dataset`).

    Returns:
        The word index dictionary.
    """
    path = get_file(
        path,
        origin='https://s3.amazonaws.com/text-datasets/imdb_word_index.
 json')
    f = open(path)
    data = json.load(f)
    f.close()
    return data
```

load_cifar10.py

```
# Copyright 2015 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
```

```
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================================
"""CIFAR10 small image classification dataset.
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os

import numpy as np

from tensorflow.python.keras._impl.keras import backend as K
from tensorflow.python.keras._impl.keras.datasets.cifar import load_batch
from tensorflow.python.keras._impl.keras.utils.data_utils import get_file


def load_data():
  """Loads CIFAR10 dataset.

  Returns:
      Tuple of Numpy arrays: `(x_train, y_train), (x_test, y_test)`.
  """
  dirname = 'cifar-10-batches-py'
  origin = 'https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz'
  path = get_file(dirname, origin=origin, untar=True)

  num_train_samples = 50000

  x_train = np.empty((num_train_samples, 3, 32, 32), dtype='uint8')
  y_train = np.empty((num_train_samples,), dtype='uint8')

  for i in range(1, 6):
    fpath = os.path.join(path, 'data_batch_' + str(i))
    (x_train[(i - 1) * 10000:i * 10000, :, :, :],
     y_train[(i - 1) * 10000:i * 10000]) = load_batch(fpath)

  fpath = os.path.join(path, 'test_batch')
  x_test, y_test = load_batch(fpath)

  y_train = np.reshape(y_train, (len(y_train), 1))
  y_test = np.reshape(y_test, (len(y_test), 1))

  if K.image_data_format() == 'channels_last':
    x_train = x_train.transpose(0, 2, 3, 1)
    x_test = x_test.transpose(0, 2, 3, 1)

  return (x_train, y_train), (x_test, y_test)
```

ExQI9.f

```fortran
module TensProduct
implicit none

contains
!       Print real part of a complex matrix on a file
subroutine MatPrintCR(A, nr, nc, filename, extra, message)
!           Scalar variables
            integer :: nr, nc
            character :: extra !To print extra information
!           Array variables
            complex, dimension(:,:), allocatable :: A
            character(:), allocatable :: filename
            character(:), allocatable, intent(in), optional :: message
!           Local scalars
            integer :: ii, jj
            real :: norm
!           Local array
            character(:), allocatable :: loc_mes, def_mes

            def_mes='    '
            if (present(message)) then
                loc_mes = message
            else
                loc_mes = def_mes
            end if

            open(10, FILE=filename, STATUS='unknown', ACCESS='append')
            if(extra=='Y') write(10,*)  loc_mes
            if(extra=='Y') write(10,*) 'Dimensions: (', nr, ',', nc, ')'
            do  ii=1,nr
                write(10, *) (real(A(ii,jj)), jj=1,nc)
            end do
            if(extra=='Y') write(10,*) '     '
            close(10)
end subroutine

!       Print a real vector as row on a file
subroutine VecPrintR(a, nn, filename, extra, message)
!           Scalar variables
            integer :: nn
            character :: extra !To print extra information
!           Array variables
            real, dimension(:), allocatable :: a
            character(:), allocatable :: filename
            character(:), allocatable, intent(in), optional :: message
!           Local scalars
            integer :: ii
!           Local array
            character(:), allocatable :: loc_mes, def_mes

            def_mes='    '
            if (present(message)) then
                loc_mes = message
            else
                loc_mes = def_mes
            end if

            open(10, FILE=filename, STATUS='unknown', ACCESS='append')
```

```fortran
            if(extra=='Y') write(10,*)  loc_mes
            if(extra=='Y')write(10,*) 'Dimension: ', nn
            write(10, *) (a(ii), ii=1,nn)
            if(extra=='Y') write(10,*) '     '
            close(10)
        end subroutine

!       Performs the tensor product between two matices A(x)B
        subroutine TensProd(A, ra, ca, B, rb, cb, C)
!           Scalar variables
            integer :: ra, ca, rb, cb
!           Array variables
            complex, dimension(:,:), allocatable :: A, B, C
!           Local scalars
            integer :: ii, jj, kk, ll

            allocate(C(ra*rb,ca*cb))
            do ii=1,rb
                do jj=1,cb
                    do kk=1,ra
                        do ll=1,ca
                        C(ra*(ii-1)+kk,ca*(jj-1)+ll)=A(kk,ll)*B(ii,jj)
                        end do
                    end do
                end do
            end do
        end subroutine

!       Performs the tensor product with identity on right
        subroutine TensProdIDR(A, ra, ca, NID, C)
!           Scalar variables
            integer :: ra, ca, NID
!           Array variables
            complex, dimension(:,:), allocatable :: A, C
!           Local scalars
            integer :: ii, kk, ll

            allocate(C(ra*NID,ca*NID))
            C=0.0
            do ii=1,NID
                do kk=1,ra
                    do ll=1,ca
                    C(ra*(ii-1)+kk,ca*(ii-1)+ll)=A(kk,ll)
                    end do
                end do
            end do
        end subroutine

!       Performs the tensor product with identity on left
        subroutine TensProdIDL(NID, B, rb, cb, C)
!           Scalar variables
            integer :: NID, rb, cb
!           Array variables
            complex, dimension(:,:), allocatable :: B, C
!           Local scalars
            integer :: ii, jj, kk, ll

            allocate(C(NID*rb,NID*cb))
```

```fortran
            C=0.0
            do ii=1,rb
                do jj=1,cb
                    do kk=1,NID
                        C(NID*(ii-1)+kk,NID*(jj-1)+kk)=B(ii,jj)
                    end do
                end do
            end do
        end subroutine

!       Swap MNEW and MOLD
        subroutine SwapON(MOLD, MNEW, ii)
!           Array variables
            complex, dimension(:,:), allocatable :: MOLD, MNEW
!           Scalar variables
            integer :: ii

            deallocate(MOLD)
            allocate(MOLD(2**ii,2**ii))
            MOLD=MNEW
            deallocate(MNEW)
        end subroutine

        end module



        program IsingModel
        use TensProduct
        implicit none
!       Scalar variables
        integer :: NN, IS, N2, KK
        real :: lambda
!       Array variables
        complex, dimension(:,:), allocatable :: ID2, SX, SZ, HN
        real, dimension(:), allocatable :: ev
!       Local scalars
        integer :: ii, info1, lwork
        logical :: DB, DB2
!       Local arrays
        complex, dimension(:,:), allocatable :: MOLD, MNEW
        character(:), allocatable :: checkfile, mex1, mex2
        complex, dimension(:), allocatable :: work
        real, dimension(:), allocatable :: rwork


!       Initilaize debug valiables
        DB=.false.
        checkfile='checks.txt'
        mex1='Hamiltonian matrix'
        mex2='Hamiltonian matrix (diagonalized)'

!       Initialize Pauli matrices and 2x2 Identity
        allocate(ID2(2,2))
        allocate(SX(2,2))
        allocate(SZ(2,2))
        ID2=0.0
        ID2(1,1)=1.0
```

```fortran
        ID2(2,2)=1.0
        SX=0.0
        SX(1,2)=1.0
        SX(2,1)=1.0
        SZ=0.0
        SZ(1,1)=1.0
        SZ(2,2)=-1.0

!       Open input file and read parameters
        open(77, FILE='parameters.txt', STATUS='old')
        read(77,*) NN, lambda, KK
        close (77, STATUS='keep')

!       Initialize hamiltonian matrix
        N2=2**NN
        allocate(HN(N2, N2))
        HN=0.0

!       First step external field term
        allocate(MOLD(2,2))
        MOLD=SZ
        do ii=2,NN
            call TensProdIDR(MOLD, 2**(ii-1), 2**(ii-1), 2, MNEW)
            call SwapON(MOLD, MNEW, ii)
        end do
        HN=HN+MOLD
        deallocate(MOLD)

!       Recursive steps external field term
        do IS=2, NN
            allocate(MOLD(2,2))
            MOLD=ID2
            do ii=2,IS-1
                call TensProdIDL(2, MOLD, 2**(ii-1), 2**(ii-1), MNEW)
                call SwapON(MOLD, MNEW, ii)
            end do
            call TensProd(MOLD, 2**(IS-1), 2**(IS-1), SZ, 2, 2, MNEW)
            call SwapON(MOLD, MNEW, IS)
            do ii=IS+1,NN
                call TensProdIDR(MOLD, 2**(ii-1), 2**(ii-1), 2, MNEW)
                call SwapON(MOLD, MNEW, ii)
            end do
            HN=HN+MOLD
            deallocate(MOLD)
        end do

!       First step coupling term
        call TensProd(SX, 2, 2, SX, 2, 2, MOLD)
        do ii=3,NN
            call TensProdIDR(MOLD, 2**(ii-1), 2**(ii-1), 2, MNEW)
            call SwapON(MOLD, MNEW, ii)
        end do
        HN=HN+lambda*MOLD
        deallocate(MOLD)

!       Recursive steps coupling term
        do IS=2, NN-1
            allocate(MOLD(2,2))
```

```
            MOLD=ID2
            do ii=2,IS-1
                call TensProdIDL(2, MOLD, 2**(ii-1), 2**(ii-1), MNEW)
                call SwapON(MOLD, MNEW, ii)
            end do
            call TensProd(MOLD, 2**(IS-1), 2**(IS-1), SX, 2, 2, MNEW)
            call SwapON(MOLD, MNEW, IS)
            call TensProd(MOLD, 2**IS, 2**IS, SX, 2, 2, MNEW)
            call SwapON(MOLD, MNEW, IS)
            do ii=IS+2,NN
                call TensProdIDR(MOLD, 2**(ii-1), 2**(ii-1), 2, MNEW)
                call SwapON(MOLD, MNEW, ii)
            end do
            HN=HN+lambda*MOLD
            deallocate(MOLD)
        end do
        if(DB) call MatPrintCR(HN, N2, N2, checkfile, 'Y', mex1)

!       Diagonalize the hamiltonian matrix and store the eigenvalues
        allocate(ev(N2))
        lwork=2*(N2)
        allocate(work(max(1,lwork)))
        allocate(rwork(max(1,3*N2-2)))
        call cheev('V', 'U', N2, HN, N2, ev, work, lwork, rwork, info1)
        if(DB) call MatPrintCR(HN, N2, N2, checkfile, 'Y', mex2)

!       Save the ground state evec and eval on file (binary format)
        open(10, FORM='unformatted',
     &           FILE='evec.bin', STATUS='unknown', ACCESS='append')
        write(10) complex(lambda,0.0), (HN(ii,1), ii=1,N2)
        close(10)

        if(DB) then
        open(10, FILE='evec.txt', STATUS='unknown', ACCESS='append')
        write(10, *) complex(lambda,0.0), (HN(ii,1), ii=1,N2)
        close(10)
        end if

        open(10, FORM='unformatted',
     &           FILE='eval.bin', STATUS='unknown', ACCESS='append')
        write(10) lambda, ev(1)
        close(10)

!       Free memory
        deallocate(HN)
        deallocate(ev)

        end program
```

wfgen.py

```
import numpy as np
import time

# Performs tensorproduct between two vectors c=a(x)b
def TensProd(a, b):
    c=np.zeros(len(a)*len(b), dtype=complex)
    for ii in range(len(b)):
        c[ii*len(a):(ii*len(a)+len(a))]=b[ii]*a[:]
```

```python
    return c

# Generates a random normalized wavefunction
def RandomWF(DD):
    psi=np.random.normal(size=DD)+np.random.normal(size=DD)*1j
    psi=psi/np.linalg.norm(psi)
    return psi


DD=2 # Hilbert space size
Npart=10     # Num of subparts
Nsep=4000 # Num of sep wf to campionate
Nnsep=4000 # Num of entangled wf to campionate

# Separable wf generation
t0=time.time()

a=RandomWF(DD)
for jj in range(Npart-1):
    b=RandomWF(DD)
    a=TensProd(a, b)

sep=a.reshape(1,-1)
for ii in range(Nsep-1):
    a=RandomWF(DD)
    for jj in range(Npart-1):
        b=RandomWF(DD)
        a=TensProd(a, b)
    sep=np.concatenate((sep,a.reshape(1,-1)), axis=0)
print('Generated an array of shape: {}').format(sep.shape)

sepRE=np.real(sep).reshape(Nsep, DD**Npart, 1)
sepIM=np.imag(sep).reshape(Nsep, DD**Npart, 1)
np.save('sep.npy', np.concatenate((sepRE,sepIM), axis=2))

t1=time.time()
print('Time to generate separable wf: '+str(t1-t0))

# Entangled wf generation
t0=time.time()

nsep=RandomWF(DD**Npart).reshape(1,-1)
for ii in range(Nnsep-1):
    a=RandomWF(DD**Npart).reshape(1,-1)
    nsep=np.concatenate((nsep,a), axis=0)
print('Generated an array of shape: {}').format(nsep.shape)

nsepRE=np.real(nsep).reshape(Nnsep, DD**Npart, 1)
nsepIM=np.imag(nsep).reshape(Nnsep, DD**Npart, 1)
np.save('nsep.npy', np.concatenate((nsepRE,nsepIM), axis=2))

t1=time.time()
print('Time to generate entangled wf: '+str(t1-t0))
```