Deep Learning Basics (lecture notes)

Romain Tavenard

Sep 26, 2022

CONTENTS

by Romain Tavenard

This document serves as lecture notes for a course that is taught at Université de Rennes 2 (France) and EDHEC Lille (France).

The course deals with the basics of neural networks for classification and regression over tabular data (including optimization algorithms for multi-layer perceptrons), convolutional neural networks for image classification (including notions of transfer learning) and sequence classification / forecasting.

The labs for this course will use `keras`, hence so will these lecture notes.

0.1 Introduction

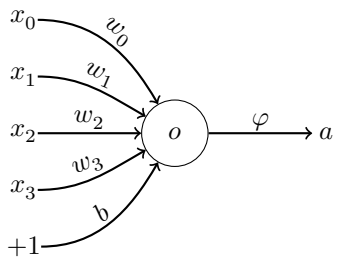
In this introduction chapter, we will present a first neural network called the Perceptron. This model is a neural network made of a single neuron, and we will use it here as a way to introduce key concepts that we will detail later in the course.

0.1.1 The model

In the neural network terminology, a neuron is a parametrized function that takes a vector $\mathbf{x}$ as input and outputs a single value a as follows:

$$a = \varphi(\underbrace{\mathbf{w}\mathbf{x} + b}_{o}),$$

where the parameters of the neuron are its weights stored in $\mathbf{w}$ and a bias term b, and φ is an activation function that is chosen *a priori* (we will come back to it in more details later in the course):



A model made of a single neuron is called a Perceptron.

0.1.2 Optimization

The models presented in this book are aimed at solving prediction problems, in which the goal is to find "good enough" parameter values for the model at stake given some observed data.

The problem of finding such parameter values is coined optimization and the deep learning field makes extensive use of a specific family of optimization strategies called **gradient descent**.

Gradient Descent

To make one's mind about gradient descent, let us assume we are given the following dataset about house prices:

```
import pandas as pd

boston = pd.read_csv("data/boston.csv")[["RM", "PRICE"]]
boston
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In [2], line 1
----> 1 import pandas as pd
      3 boston = pd.read_csv("data/boston.csv")[["RM", "PRICE"]]
      4 boston

ModuleNotFoundError: No module named 'pandas'
```

In our case, we will try (for a start) to predict the target value of this dataset, which is the median value of owner-occupied homes in $1000 `"PRICE"`, as a function of the average number of rooms per dwelling `"RM"` :

```
x = boston["RM"]
y = boston["PRICE"]

plt.scatter(x, y);
```

A short note on this model

In the Perceptron terminology, this model:

- has no activation function (*i.e.* φ is the identity function)
- has no bias (*i.e.* b is forced to be 0, it is not learnt)

Let us assume we have a naive approach in which our prediction model is linear without intercept, that is, for a given input x_i the predicted output is computed as:

$$\hat{y_i} = wx_i$$

where w is the only parameter of our model.

Let us further assume that the quantity we aim at minimizing (our objective, also called loss) is:

$$\mathcal{L}(w) = \sum_i (\hat{y_i} - y_i)^2$$

where y_i is the ground truth value associated with the i-th sample in our dataset.

Let us have a look at this quantity as a function of w:

```
import numpy as np

def loss(w, x, y):
    w = np.array(w)
    return np.sum(
```

(continues on next page)

(continued from previous page)

```
        (w[:, None] * x[None, :] - y[None, :]) ** 2,
        axis=1
    )

w = np.linspace(-2, 10, num=100)

plt.plot(w, loss(w, x, y), "r-");
```

Here, it seems that a value of w around 4 should be a good pick, but this method (generating lots of values for the parameter and computing the loss for each value) cannot scale to models that have lots of parameters, so we will try something else.

Let us suppose we have access, each time we pick a candidate value for w, to both the loss $\mathcal{L}$ and information about how $\mathcal{L}$ varies, locally. We could, in this case, compute a new candidate value for w by moving from the previous candidate value in the direction of steepest descent. This is the basic idea behind the gradient descent algorithm that, from an initial candidate w_0, iteratively computes new candidates as:

$$w_{t+1} = w_t - \rho \left. \frac{\partial \mathcal{L}}{\partial w} \right|_{w=w_t}$$

where ρ is a hyper-parameter (called the learning rate) that controls the size of the steps to be done, and $\left. \frac{\partial \mathcal{L}}{\partial w} \right|_{w=w_t}$ is the gradient of $\mathcal{L}$ with respect to w, evaluated at $w = w_t$. As you can see, the direction of steepest descent is the opposite of the direction pointed by the gradient (and this holds when dealing with vector parameters too).

This process is repeated until convergence, as illustrated in the following visualization:

```
rho = 1e-5

def grad_loss(w_t, x, y):
    return np.sum(
        2 * (w_t * x - y) * x
    )


ww = np.linspace(-2, 10, num=100)
plt.plot(ww, loss(ww, x, y), "r-", alpha=.5);

w = [0.]
for t in range(10):
    w_update = w[t] - rho * grad_loss(w[t], x, y)
    w.append(w_update)

plt.plot(w, loss(w, x, y), "ko-");
```

What would we get if we used a smaller learning rate?

```
rho = 1e-6

ww = np.linspace(-2, 10, num=100)
plt.plot(ww, loss(ww, x, y), "r-", alpha=.5);

w = [0.]
for t in range(10):
    w_update = w[t] - rho * grad_loss(w[t], x, y)
    w.append(w_update)

plt.plot(w, loss(w, x, y), "ko-");
```

It would definitely take more time to converge. But, take care, a larger learning rate is not always a good idea:

```
rho = 5e-5

ww = np.linspace(-2, 10, num=100)
plt.plot(ww, loss(ww, x, y), "r-", alpha=.5);

w = [0.]
for t in range(10):
    w_update = w[t] - rho * grad_loss(w[t], x, y)
    w.append(w_update)

plt.plot(w, loss(w, x, y), "ko-");
```

See how we are slowly diverging because our steps are too large?

0.1.3 Using the perceptron for supervised classification

TODO: present the notions of activation function and logistic loss

0.1.4 Wrap-up

In this section, we have introduced:

- a very simple model, called the Perceptron: this will be a building block for the more advanced models we will detail later in the course, such as:
  - the *Multi-Layer Perceptron*
  - *Convolutional architectures*
  - *Recurrent architectures*
- the fact that a task comes with a loss function to be minimized (here, we have used the *mean squared error (MSE)* for regression and *logistic loss* for classification);
- the concept of gradient descent to optimize the chosen loss over a model's single parameter, which will be extended in *a dedicated chapter*.

0.2 Multi Layer Perceptrons (MLP)

0.3 Optimization

In this chapter, we will present an optimization strategy called **Gradient Descent** and its variants, and show how they can be used to optimize neural network parameters.

0.3.1 SGD

0.3.2 Variants of SGD (towards Adam)

0.3.3 The curse of depth

0.4 Convolutional Neural Networks (CNN)

0.4.1 CNNs *à la* LeNet

0.4.2 Anatomy of a ResNet

0.4.3 Using a pre-trained model for better performance

0.4.4 ConvNets for time series

Convolutional neural networks for time series rely on the 1d convolution operator that, given a time series $\mathbf{x}$ and a filter $\mathbf{f}$, computes an activation map as:

$$(\mathbf{x} * \mathbf{f})(t) = \sum_{k=1}^{L} f_k x_{t+k} \tag{1}$$

where L is the length (number of timestamps) of the filter $\mathbf{f}$.

The following code illustrates this notion using a Gaussian filter:

```
import numpy as np

def random_walk(size):
    rnd = np.random.randn(size) * .1
    ts = rnd
    for t in range(1, size):
        ts[t] += ts[t - 1]
    return ts

np.random.seed(0)
x = random_walk(size=50)
f = np.exp(- np.linspace(-2, 2, num=5) ** 2 / 2)
f /= f.sum()

plt.figure()
plt.plot(x, label='raw time series')
plt.plot(np.correlate(x, f, mode='same'),
         label='activation map (gaussian smoothed time series)')
plt.legend();
```

```
<Figure size 640x480 with 1 Axes>
```

Convolutional neural networks are made of convolution blocks whose parameters are the coefficients of the filters they embed (hence filters are not fixed *a priori* as in the example above but rather learned). These convolution blocks are translation equivariant, which means that a (temporal) shift in their input results in the same temporal shift in the output:

```
/tmp/ipykernel_4919/368849627.py:32: UserWarning: This figure includes Axes that␣
↪are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
/tmp/ipykernel_4919/368849627.py:23: MatplotlibDeprecationWarning: Auto-removal of␣
↪overlapping axes is deprecated since 3.6 and will be removed two minor releases␣
↪later; explicitly call ax.remove() as needed.
  fig2 = plt.subplot(2, 1, 2)
```

```
<IPython.core.display.HTML object>
```

As we have seen earlier, convolutional models are known to perform very well in computer vision applications, using moderate amounts of parameters compared to fully connected ones (of course, counter-examples exist, and the term "moderate" is especially vague).

Most standard time series architectures that rely on convolutional blocks are straight-forward adaptations of models from the computer vision community ([Le Guennec *et al.*, 2016] relies on an old-fashioned alternance between convolution and pooling layers, while more recent works rely on residual connections and inception modules [Fawaz *et al.*, 2020]).

These models (and more) are presented and benchmarked in [Fawaz *et al.*, 2019] that we advise the interested reader to refer to for more details.

0.4.5 References

0.5 Recurrent Neural Networks

Recurrent neural networks (RNNs) proceed by processing elements of a time series one at a time. Typically, at time t, a recurrent block will take both the current input x_t and a hidden state h_{t-1} that aims at summarizing the key information from past inputs $\{x_0, \dots, x_{t-1}\}$, and will output an updated hidden state h_t. There exist various recurrent modules that mostly differ in the way h_t is computed.

0.5.1 "Vanilla" RNNs

The basic formulation for a RNN block is as follows:

$$\forall t, h_t = \tanh(W_h h_{t-1} + W_x x_t + b) \tag{1}$$

where W_h is a weight matrix associated to the processing of the previous hidden state, W_x is another weight matrix associated to the processing of the current input and b is a bias term.

Note here that W_h, W_x and b are not indexed by t, which means that they are **shared across all timestamps**.

An important limitation of this formula is that it easily fails at capturing long-term dependencies. To better understand why, one should remind that the parameters of these networks are optimized through stochastic gradient descent algorithms.

To simplify notations, let us consider a simplified case in which h_t and x_t are both scalar values, and let us have a look at what the actual gradient of the output h_t is, with respect to W_h (which is then also a scalar):

$$\nabla_{W_h}(h_t) = \tanh'(o_t) \cdot \frac{\partial o_t}{\partial W_h} \tag{2}$$

where $o_t = W_h h_{t-1} + W_x x_t + b$, hence:

$$\frac{\partial o_t}{\partial W_h} = h_{t-1} + W_h \cdot \frac{\partial h_{t-1}}{\partial W_h}\,. \tag{3}$$

Here, the form of $\frac{\partial h_{t-1}}{\partial W_h}$ will be similar to that of $\nabla_{W_h}(h_t)$ above, and, in the end, one gets:

$$\begin{aligned}
\nabla_{W_h}(h_t) &= \tanh'(o_t) \cdot \left[h_{t-1} + W_h \cdot \frac{\partial h_{t-1}}{\partial W_h} \right] && (4)\\
&= \tanh'(o_t) \cdot \left[h_{t-1} + W_h \cdot \tanh'(o_{t-1}) \cdot [h_{t-2} + W_h \cdot [...]] \right] && (5)\\
&= h_{t-1}\tanh'(o_t) + h_{t-2}W_h\tanh'(o_t)\tanh'(o_{t-1}) + ... && (6)\\
&= \sum_{t'=1}^{t-1} h_{t'} \left[W_h^{t-t'-1}\tanh'(o_{t'+1}) \cdot \cdots \cdot \tanh'(o_t) \right] && (7)
\end{aligned}$$

In other words, the influence of $h_{t'}$ will be mitigated by a factor $W_h^{t-t'-1}\tanh'(o_{t'+1}) \cdot \cdots \cdot \tanh'(o_t)$.

Now recall what the tanh function and its derivative look like:

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In [2], line 1
----> 1 import torch
      3 def tanh(x):
      4     return 2. / (1. + torch.exp(-2 * x)) - 1.

ModuleNotFoundError: No module named 'torch'
```

One can see how quickly gradients gets close to 0 for inputs larger (in absolute value) than 2, and having multiple such terms in a computation chain will likely make the corresponding terms vanish.

In other words, the gradient of the hidden state at time t will only be influenced by a few of its predecessors $\{h_{t-1}, h_{t-2}, ...\}$ and long-term dependencies will be ignored when updating model parameters through gradient descent. This is an occurrence of a more general phenomenon known as the **vanishing gradient** effect.

0.5.2 Long Short-Term Memory

The Long Short-Term Memory (LSTM, [Hochreiter and Schmidhuber, 1997]) blocks have been designed as an alternative recurrent block that aims at mitigating this vanishing gradient effect through the use of gates that explicitly encode pieces of information that should (resp. should not) be kept in computations.

Gates in neural networks

In the neural networks terminology, a gate $g \in [0, 1]^d$ is a vector that is used to filter out information from an incoming feature vector $v \in \mathbb{R}^d$ such that the result of applying the gate is: $g \odot v$ where $\odot$ is the element-wise product. The gate g will hence tend to remove part of the features in v (those corresponding to very low values in g).

In these blocks, an extra state is used, referred to as the cell state C_t. This state is computed as:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad (8)$$

where f_t is the forget gate (which pushes the network to forget about useless parts of the past cell state), i_t is the input gate and $\tilde{C}_t$ is an updated version of the cell state (which, in turn, can be partly censored by the input gate).

Let us delay for now the details about how these 3 terms are computed, and rather focus on how the formula above is significantly different from the update rule of the hidden state in vanilla RNNs. Indeed, in this case, if the network learns so (through f_t), the full information from the previous cell state C_{t-1} can be recovered, which would allow gradients to flow through time (and not vanish anymore).

Then, the link between the cell and hidden states is:

$$h_t = o_t \odot \tanh(C_t)\,. \tag{9}$$

In words, the hidden state is the tanh-transformed version of the cell state, further censored by an output gate o_t.

All gates used in the formulas above are defined similarly:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{10}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{11}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{12}$$

where σ is the sigmoid activation function (which has values in $[0, 1]$) and $[h_{t-1}, x_t]$ is the concatenation of h_{t-1} and x_t features.

Finally, the updated cell state $\tilde{C}_t$ is computed as:

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)\,. \tag{13}$$

Many variants over these LSTM blocks exist in the literature that still rely on the same basic principles.

0.5.3 Gated Recurrent Unit

A slightly different parametrization of a recurrent block is used in the so-called Gatted Recurrent Unit (GRU, [Cho *et al.*, 2014]).

GRUs also rely on the use of gates to (adaptively) let information flow through time. A first significant difference between GRUs and LSTMs, though, is that GRUs do not resort to the use of a cell state. Instead, the update rule for the hidden state is:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{14}$$

where z_t is a gate that balances (per feature) the amount of information that is kept from the previous hidden state with the amount of information that should be updated using the new candidate hidden state $\tilde{h}_t$, computed as:

$$\tilde{h}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t] + b)\,, \tag{15}$$

where r_t is an extra gate that can hide part of the previous hidden state.

Formulas for gates z_t and r_t are similar to those provided for f_t, i_t and o_t in the case of LSTMs.

A study of the ability of these variants of recurrent networks to learn long-term dependencies is provided in this online publication.

0.5.4 Conclusion

In this chapter, we have reviewed neural network architectures that are used to learn from time series datasets. Because of time constraints, we have not tackled attention-based models in this course. We have presented convolutional models that aim at extracting discriminative local shapes in the series and recurrent models that rather leverage the notion of sequence. Concerning the latter, variants that aim at facing the vanishing gradient effect have been introduced. Note that recurrent models are known to require more training data than their convolutional counterparts in order to learn meaningful representations.

0.5.5 References

BIBLIOGRAPHY

[FFW+19] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33(4):917–963, 2019.

[FLF+20] Hassan Ismail Fawaz, Benjamin Lucas, Germain Forestier, Charlotte Pelletier, Daniel F Schmidt, Jonathan Weber, Geoffrey I Webb, Lhassane Idoumghar, Pierre-Alain Muller, and François Petitjean. Inceptiontime: finding alexnet for time series classification. *Data Mining and Knowledge Discovery*, 34(6):1936–1962, 2020.

[LGMT16] Arthur Le Guennec, Simon Malinowski, and Romain Tavenard. Data Augmentation for Time Series Classification using Convolutional Neural Networks. In *ECML/PKDD Workshop on Advanced Analytics and Learning on Temporal Data*. Riva Del Garda, Italy, September 2016.

[CVMerrienboerBB14] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: encoder-decoder approaches. 2014. arXiv:1409.1259.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.