Apprentissage Statistique Automatique II

Lucas Reding, Thibault Defourneau

Lecture note

Contents

1 Introduction

Deep learning is a set of learning methods attempting to model data with complex architectures combining different non-linear transformations. The elementary bricks of deep learning are the neural networks, that are combined to form the deep neural networks.

These techniques have enabled significant progress in the fields of sound and image processing, including facial recognition, speech recognition, computer vision, automated language processing, text classification (for example spam recognition). Potential applications are very numerous. A spectacularly example is the AlphaGo program, which learned to play the go game by the deep learning method, and beated the world champion in 2016.

There exist several types of architectures for neural networks :

- The multilayer perceptrons, that are the oldest and simplest ones
- The recurrent neural networks, used for sequential data such as text or times series.
- The Convolutional Neural Networks (CNN), particularly adapted for image processing
- The LLMs, which now widely used in various applications

They are based on deep cascade of layers. They need clever stochastic optimization algorithms, and initialization, and also a clever choice of the structure. They lead to very impressive results, although very few theoretical fondations are available till now.

2 Tensorflow

2.1 Basics

Tensorflow is a framework developped by Google Brain which has been designed for Machine learning. It is the core tool behind some well-known services like Google Cloud Speech, Google Photos and Google Search. In the following lines of code, we present the main component of this framework:

```
import tensorflow as tf

# High level of Deep learning API
tf.keras

# Low level of Deep learning API
tf.nn

# Non-exhaustive list of numpy like packages
tf.math
tf.linalg
tf.signal
tf.random
tf.bitwise

# Data processing
tf.data
tf.audio
tf.images
tf.io
tf.queue

# Optimization and Deployment
tf.distribute
tf.saved_model
tf.autograph
tf.graph_util
tf.lite
tf.quantization
tf.tpu

# Special data structure
tf.lookup
tf.nest
tf.ragged
tf.sets
tf.sparse
tf.strings
```

Remark 1. *In the lowest level, each Tensorflow operation is implemented in C++. Moreover, may operations have several implementations, called kernels. Each kernel is dedicated to a specific type of processor like CPU, GPU or*

even TPU (Tensor Processing Unit).

Using Tensorflow is a similar experience than using the package Numpy. However the naming of basic operations may differ. We are now comparing some similar methods between Tensorflow and Numpy:

```
import numpy as np
import tensorflow as tf

# Matrix definition
tensor = tf.constant([[1, 2, 3], [4, 5, 6]]) # in Tensorflow
arr = np.array([[1, 2, 3], [4, 5, 6]]) # in Numpy
arr_from_tensor = tensor.numpy() # It is possible to retrieve the numpy version of a tensor

# Matrix multiplication
tensor @ tf.transpose(tensor) # in Tensorflow
arr.dot(arr.T) # in Numpy

# Mean computation
tf.reduce_mean(tensor) # in Tensorflow
np.sum(arr) # in Numpy
```

Remark 2. *In the above examples, the variable tensor is a $tf.Tensor$ object which is immutable. However, we need an object which can be modified in some context, as during backpropagation step. This is why the object $tf.Variable$ exists.*

```
import numpy as np
import tensorflow as tf

var = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
arr = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])

var.assign(2 * v) # In Tensor, it replaces var with 2 * var using assign method
arr = 2 * arr # In numpy, we keep the affectation '='
```

Exercises 1. *Write the Huber loss using Tensorflow using only the basic operations. The Huber loss is defined as follows:*

$$Huber(x) = \begin{cases} \frac{1}{2}\,x^2 & , if\,|x| \leq 1 \\ |x| - \frac{1}{2} & , if\,|x| > 1 \end{cases}$$

2.2 Loss wrapper

Defining a custom loss is possible by using the base class $tf.keras.losses.Loss$, with two methods to customize:

- The method *call* gets the targets and the predictions for calculating all the losses during the training step;
- The method get_config returns a dictionary which associates each hyperparameter name to its value.

Let's define the Huber loss with this approach:

```
import tensorflow as tf

class HuberLoss(tf.keras.losses.Loss):
    def __init__(self, threshold = 1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)

    def call(self, y_true, y_pred):
        # The logic already defined in the first exercise
        return computed_loss

    def get_config(self):
        base_config = super().get_config()
        return {**bas_config, "threshold": self.threshold}
```

This way of defining a custom loss is very practical, because all necessary abstractions have already been taken into account in the base class $tf.keras.losses.Loss$. In particular, it is easy to save the hyperparameters of the custom loss function when saving the trained model:

```python
import tensorflow as tf

# Let `model` be any Tensorflow model. When compiling it with the loss function, it is enought to write:
model.compile(loss=HuberLoss(2,), optimizer="nadam")

# When loading the trained model, it is important to specify the custom loss as well:
model = tf.keras.models.load_model("my_model_with_the_custom_huber_loss", custom_objects={"HuberLoss": HuberLoss})
```

2.3 Regularizer wrapper

Similarly to the custom loss, it is possible to define a custom regularizer by using the base class $tf.keras.regularizers.Regularizer$. Let's take an example by defining the l_1 regularization:

```python
import tensorflow as tf

class HuberLoss(tf.keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))

    def get_config(self):
        return {"factor": self.factor}
```

Remark 3. *As we can observe, there are some differences between a custom loss and a custom regularizer:*

- *We don't need to invoke the parent builder, as well as the parent configuration.*
- *We need to define the __call__ method in the class method instead of call method.*

2.4 Metric wrapper

```python
import tensorflow as tf

class HuberMetric(tf.keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs)
        self.threshold = threshold
        self.huber_fn = create_huber_(threshold)
        self.total = self.add_weigth("total", initializer="zeros")
        self.count = self.add_weigth("count", initializer="zeros")

    def update_state(self, y_true, y_pred, sample_weight=None):
        sample_metrics = self.huber_fun(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(sample_weights))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))

    def result(self):
        return self.total / self.count

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

Let's explain the above code:

- The builder $tf.keras.metrics.Metric$ uses the method add_weight in order to generate variables for computing the metric over several batches.
- The method *update_state* updates the variables from targets and predictions from a single batch.
- The method *result* provides the final output, which is here the mean of Huber metric over all batches.
- The builder also provides the method *reset_states* which initializes again all the variables to 0.0. It can be customized if necessary.

2.5 Custom layer with Keras

```python
import tensorflow as tf

class CustomDenseLayer(tf.keras.layers.Layer):
    def __init__(self, units=None, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = tf.keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(name="kernel", shape=[batch_input_shape[-1], self.units],
        ↪  initializer="glorot_normal")
        self.biais = self.add_weight(name="biais", shape=[self.units], initializer="zeros")

    def call(self):
        return self.activation(X @ self.kernel + self.biais)

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units, "activation": tf.keras.activations.serialize(self.activation)}
```

Exercises 2. *Implement a custom layer performing a normalization:*

- *The method build should have two trainable weights of type* $tf.float32$ *called* α *and* β *respectively initialized to* 1 *and* 0.
- *The method call must compute the mean* μ *and the standard deviation* σ *for each instance, by using the method* $tf.nn.moments(inputs, axes = -1, keepdims = True)$. *The final output should be* $\alpha \otimes (X - \mu)/(\sigma + \epsilon) + \beta$, *where* $\otimes$ *is the pointwise multiplication and* ϵ *equals to* 0.001.
- *Check if your custom implementation produces the same result as* $tf.keras.layers.LayerNormalization$.

3 Regularization

Training an neural network is an optimization problem. Suppose that, for a given set of parameters W embedded into an euclidean space, we have a neural network (hence a specific non-linear smooth map) $f(-;W) : \mathbb{R}^n \to \mathbb{R}^p$. For determining the parameters W living in some euclidean space, we need:

1. A set of training points $\{(x_i, y_i)\}_{1 \leq i \leq N}$;
2. A loss function $L : \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}$ which smooth at least at each x_i;

Hence the expected parameters are those minimizing the following empirical expectation:

$$W^* = argmin_W \Big(\frac{1}{N} \sum_{i=1}^{N} L\big(f(x_i; W),\, y_i\big) \Big) \tag{1}$$

Remark 4. *It is important to note that this is not a convex optimization problem in general, and therefore there is no unique solution.*

Exercises 3. *Let $W \in \mathbb{R}^{p \times n}$ be a matrix, and $f(-;W) : \mathbb{R}^n \to \mathbb{R}^p$ the linear map defined by $f(x;W) = Wx$. Imagine that we found W^* such that it minimizes the following empirical expectation of the hinge loss:*

$$\frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} max\big(0, (f(x_i;W))_j - (f(x_i;W))_{y_i} + 1\big)$$

Is W^* unique ?

Remark 5. *The main drawbacks of this training strategy is that:*

- *It may lead to an overfitting problem, i.e. the trained neural network fits too well with the training data points, so that it won't be able to predict correctly on new data points.*
- *It eventually lead to an underfitting problem, i.e. the trained model network doesn't manage to capture enough patterns from training data points.*
- *It introduces more uncertainty, i.e. variance because of wide amplitude of some weights.*

Overfitting and underfitting problems are quite common when training a deep neural networks. A way to check when such model tend to "overfit" or "underfit" is to:

1. Split the overall data into training data, validation data and test data.
2. Track the the evolution of losses on both training data and validation data.
3. Evaluate the trained model on test data.

The below figure shows you how to identify quickly these problems:

Remark 6. *During the training step, the best model can be obtained by stopping the training of the model when the loss or metric on validation data points start to increase. This is called the early stopping strategy.*

A strategy to solve the overfitting issue is to implement a regularization strategy. This can be done following three main approaches:

1. Adding a regularization term to the optimization problem when minimizing the empirical expected loss.
2. Reducing the number of parameters in the neural network, for instance by using *Dropout*.
3. Transforming the data before or during the training process.

3.1 Regularization term

Adding a regularization term is the most common strategy for preventing from overfitting. Concretely, instead of solving (1), we try to solve the following optimization problem:

$$\frac{1}{N} \sum_{i=1}^{N} L\big(f(x_i; W),\, y_i\big) \;+\; \lambda\, R(W)\;,$$

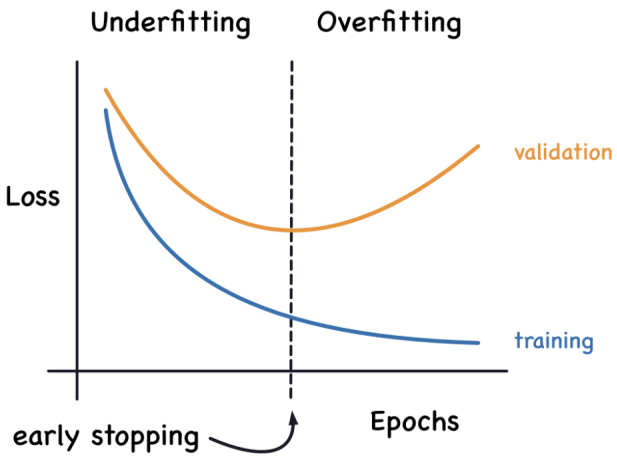

Figure 1: Evolution of either losses or metrics on training data points (in blue) and validation data point (in orange) during a training step.

where $\lambda \in \mathbb{R}$ is a parameter to optimize, and $R(W)$ is the regularization term. Here are the classical regularization terms:

- l_1-regularizer: $R(W) = ||W||_1 - C$
- l_2-regularizer: $R(W) : ||W||_2 - C$

for a fixed positive $C \in \mathbb{R}^*$.

Remark 7. *Including this regularization term ensures that the weights are bounded by C.*

Let's see a 2D visualization of this optimization problem for the loss function $L : \mathbb{R}^2 \to \mathbb{R}$ defined by:

$$\forall\, w_1, w_2 \in \mathbb{R}\,, \quad L(w_1, w_2) = (w_1 - a)^2 + (w_2 - b)^2$$

where $a = 1.5$ and $b = 1$. It is straightforward that the minimum is reached when $(w_1, w_2) = (a, b)$. We can observe that the norm of the optimal point is greater than 1. In the figure 2, we show the minimum when adding a l_1 and l_2 regularization with $C = 1$.

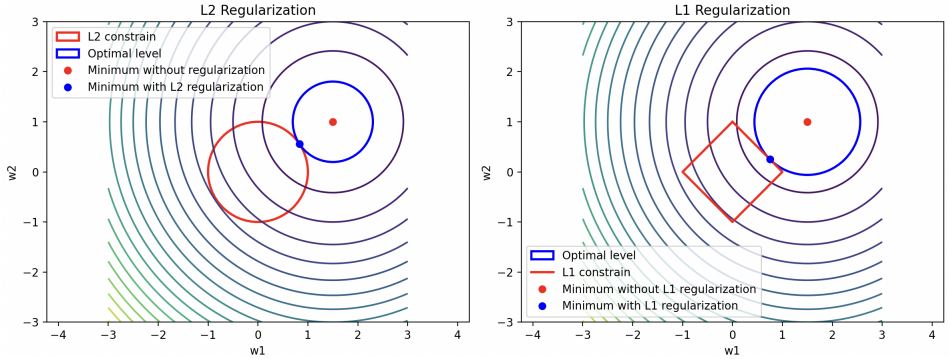

Figure 2: The contours illustrate the values of the loss function across different parameter values. The red point corresponds to the global minimum, whereas the blue point represents the minimum achieved when regularization is applied.

3.2 Dropout

The dropout is quite simple: it randomly deactivates fixed proportion of parameters from selected layers of a neural network during a training step.

Remark 8. *It is important to note that the deactivated neurons during a training step might be activated in the next one.*

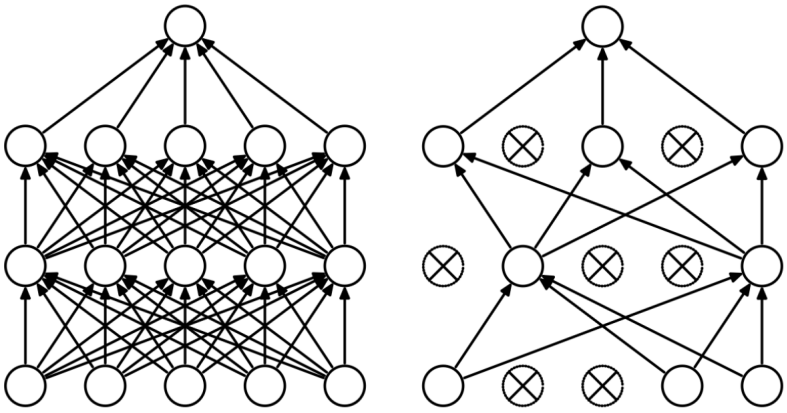

Figure 3: Example of dropout layer applied to all layers, except the last one.

The idea behind the dropout strategy is to prevent from relying too much on some neurons, which may focus on specific patterns from the inputs.

Let's illustrate the effect of a dropout on a simple model which classifies handwritten digits from the dataset MNIST provided by Tensorflow:

```
import os

import cv2

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf
import tensorflow_datasets as tfds

# Indicate a path to store your models and data

MAIN_PATH = "./dropout"

# Creation of necessary subfolders

subfolders = ["data", "models", "figures"]
for subfolder in subfolders:
    if not os.path.exists(f"{MAIN_PATH}/{subfolder}"):
        os.makedirs(f"{MAIN_PATH}/{subfolder}")


# Some utils


def normalize_img(image, label):
    """Normalizes images: `uint8` -> `float32`."""
    return tf.squeeze(tf.cast(image, tf.float32), axis=-1) / 255.0, label


def plot_losses(nb_epochs, loss, val_loss, path_to_save):
    epochs = range(1, nb_epochs + 1)

    plt.figure()

    plt.plot(epochs, loss, "bo", label="Training loss")
    plt.plot(epochs, val_loss, "b", label="Validation loss")
    plt.title("Training and validation loss")
    plt.legend()

    plt.savefig(path_to_save)
```

```
def save_test_images(ds_test, path_to_save):
    for images, labels in ds_test:
        _images = images.numpy()
        _labels = labels.numpy()

        np.save(f"{path_to_save}/labels.npy", _labels)

        for i in range(_images.shape[0]):
            _image = _images[i, :, :].reshape(28, 28)
            cv2.imwrite(f"{path_to_save}/image_{i}.png", _image)

        break


# Load the dataset

(ds_train, ds_test), ds_info = tfds.load(
    "mnist",
    split=["train", "test"],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)

ds_train = ds_train.map(normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
ds_train = ds_train.batch(32).prefetch(tf.data.AUTOTUNE)

save_test_images(
    ds_test=ds_test.batch(32).prefetch(tf.data.AUTOTUNE),
    path_to_save=f"./{MAIN_PATH}/data",
)

ds_test = ds_test.map(normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
ds_test = ds_test.batch(32).prefetch(tf.data.AUTOTUNE)

# Build the models respectively without and with the dropout

model2_without_dropout = tf.keras.Sequential(
    [
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(100, activation="relu"),
        tf.keras.layers.Dense(100, activation="relu"),
        tf.keras.layers.Dense(10, activation="softmax"),
    ]
)

model2_with_dropout = tf.keras.Sequential(
    [
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dropout(rate=0.2),
        tf.keras.layers.Dense(100, activation="relu"),
        tf.keras.layers.Dropout(rate=0.2),
        tf.keras.layers.Dense(100, activation="relu"),
        tf.keras.layers.Dropout(rate=0.2),
        tf.keras.layers.Dense(10, activation="softmax"),
    ]
)

# Indicate the training parameters

EPOCHS = 25
LR = 0.001


# Compile the models

model2_without_dropout.compile(
    optimizer=tf.keras.optimizers.Adam(LR),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)
```

```
model2_with_dropout.compile(
    optimizer=tf.keras.optimizers.Adam(LR),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
)

# Train the models

history_without_dropout = model2_without_dropout.fit(
    ds_train, validation_data=ds_test, epochs=EPOCHS
)
history_with_dropout = model2_with_dropout.fit(
    ds_train, validation_data=ds_test, epochs=EPOCHS
)

# Save the models

model2_without_dropout.save(f"./{MAIN_PATH}/models/model_without_dropout.keras")
model2_with_dropout.save(f"./{MAIN_PATH}/models/model_with_dropout.keras")

# Retrieve the losses computed during the training step

losses_training_without_dropout = history_without_dropout.history["loss"]
losses_validation_without_dropout = history_without_dropout.history["val_loss"]
losses_training_with_dropout = history_with_dropout.history["loss"]
losses_validation_with_dropout = history_with_dropout.history["val_loss"]

# Save the evolution of losses

plot_losses(
    nb_epochs=EPOCHS,
    loss=losses_training_without_dropout,
    val_loss=losses_validation_without_dropout,
    path_to_save=f"{MAIN_PATH}/figures/loss_without_dropout.png",
)

plot_losses(
    nb_epochs=EPOCHS,
    loss=losses_training_with_dropout,
    val_loss=losses_validation_with_dropout,
    path_to_save=f"{MAIN_PATH}/figures/loss_with_dropout.png",
)
```

Exercises 4. *After running the above code, answer the following questions:*

- *What is the main difference between the generated plots loss_without_dropout.png and loss_with_dropout.png ?*
- *What do you conclude ?*

3.3 Data transformation

Another common strategy for inducing implicit regularization is to transform the input data. Two common approaches are:

- **Data augmentation:** This technique artificially increases the size of the training set by creating modified versions of existing data points. In Computer Vision, for example, transformations such as rotations, flips, or color shifts help neural networks learn more robust and invariant representations.
- **Normalization layers:** These layers standardize activations by subtracting the mean and dividing by the standard deviation, typically on a per-batch basis. Normalization improves numerical stability and often accelerates training by keeping the distribution of activations more consistent across layers.

We will explore data augmentation in more detail during the exercise sessions, where you will implement and visualize several augmentation strategies and study their impact on model performance.

Normalization layers, on the other hand, primarily address the problem of *internal covariate shift*, which is the

phenomenon where the distribution of inputs to a layer changes during training as the parameters of the previous layers are updated. By keeping activations centered and scaled, normalization ensures that gradients propagate more smoothly through the network.

In the case of Batch Normalization, for a mini-batch of activations $\{x_1, \ldots, x_m\}$, the normalized output is computed as:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}},$$

where μ_B and σ_B^2 are the mean and variance computed over the mini-batch, and ε is a small constant to ensure numerical stability. The normalized values are then scaled and shifted using learnable parameters γ and β:

$$y_i = \gamma \hat{x}_i + \beta.$$

This adaptive re-scaling allows the network to preserve representational capacity while benefiting from the normalization effect.