Lecture 5: How to Make Machine Learning Itself More Explainable

1 How can we make machine learning itself more explainable: idea

What is machine learning: reminder. As we recall, machine learning means that:

- we are given examples of inputs $x^{(k)}$ and outputs $y^{(k)}$, and
- we want to find an algorithm $f(x)$ that, given the inputs $x^{(k)}$, would generate the results close to $y^{(k)}$: $f\left(x^{(k)}\right) \approx y^{(k)}$.

For example:

- we have some photos $x^{(k)}$ of pets, and
- for each photo, we have an indication $y^{(k)}$ of whether this is a cat or a dog.

We want to *train* a neural network so that,

- given a picture x,
- the network will tell whether it is a picture of cat or of a dog.

A reason why machine learning results are not explainable. One of the problems with many machine learning techniques – in particular, with deep learning – is that many aspects of these techniques come from trial and error. Researchers tried this, tried that, some thing worked, some did not. What worked is what we use now.

Is it fully convincing? Not really: maybe next year, researcher will find something which works even better.

Natural idea. The results of machine learning would be much more convincing if we had a convincing explanation of various aspects of these techniques. This is what we will study in this lecture.

2 Selection of an activation function

How neural networks work. In a neural network, signals interchangingly undergo two types of data processing procedures:

- linear transformations, when we transform the input signals $s_1, \ldots, s_n$ into their linear combination $s = a_0 + a_1 \cdot s_1 + \ldots + a_n \cdot s_n$; and
- non-linear transformations, when we transform the input signal s into an output $o = F(s)$ for some non-linear function $F(x)$.

This non-linear function $F(x)$ is known as the *activation function.*

The parameters of the corresponding linear transformations are selected so that the output of this network on given inputs $x^{(k)}$ are as close as possible to the desired output $y^{(k)}$. The determination of these parameters is known as *training.*

Traditional neural networks. In the traditional neural networks, to transform the inputs $x_1, \ldots, x_n$ into the desired output y, we apply the following three transformations:

- first, we apply several (K) linear transformations, each of which transforms the inputs $x_1, \ldots, x_n$ into their linear combination

$$y_k = w_{k0} + w_{k1} \cdot x_1 + \ldots + w_{kn} \cdot x_n;$$

- then, we apply the activation function $F(x)$ to the signals y_k, resulting in $z_k = F(y_k)$, i.e., in

$$z_k = F\left(w_{k0} + \sum_{i=1}^{n} w_{ki} \cdot x_i\right);$$

- finally, we apply a linear transformation to the values z_k to get the final result $y = W_0 + \sum\limits_{k=1}^{K} W_k \cdot z_k$, i.e.,

$$y = W_0 + \sum_{k=1}^{K} W_k \cdot F\left(w_{k0} + \sum_{i=1}^{n} w_{ki} \cdot x_i\right).$$

In the traditional neural networks, mostly, the following activation function is used – $F(x) = \dfrac{1}{1 + \exp(-x)}$. This function – known as *sigmoid* or *logistic* activation function – was selected because it adequately reflects how signals are processed in most biological neurons.

Numerical example. To illustrate how a neural network works, let us run a simple numerical example of a 2-layer neural network with two inputs $x_1 = 1$ and $x_2 = 2$.

- In the first layer, we perform a linear transformation and compute the value $y = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2$, for $w_0 = -5$, $w_1 = 1$, and $w_2 = 2$.
- In the second layer, we apply, to the result of the first layer, the sigmoid activation function $F(y) = \dfrac{1}{1 + \exp(-y)}$ and get $z = F(y)$.

What will be the result z of this data processing?

- After the first layer, we get

$$y = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = -5 + 1 \cdot 1 + 2 \cdot 2 = -5 + 1 + 4 = 0.$$

- Based on this value, on the second layer, we compute the value

$$z = f(y) = \frac{1}{1 + \exp(-y)} = \frac{1}{1 + \exp(-0)} = \frac{1}{1 + 1} = \frac{1}{2} = 0.5.$$

Deep neural networks. Interestingly, lately, a different type of neural networks turned out to be much more efficient: *deep* neural networks, in which:

- we have many layers, and
- the activation function is different: it is the function

$$F(x) = \max(0, x)$$

known as *rectified linear* function (ReLU, for short).

Natural question. A natural question is: why rectified linear activation function?

- In the traditional neural networks, the activation function was selected so as to simulate biological neurons. This makes sense: living beings are the result of billions of years of improving evolution which have perfected them.
- However, rectified linear function is used simply because it works well, there is no widely used theoretical explanation. This makes results obtained by using this function not perfectly convincing: what if someone comes up with a new activation function which works even better?

Let us try to explain why rectified linear activation function is used.

Main idea. Once the network has been training, we can use it, so that:

- given x,
- the network will generate the desired value $y = f(x)$.

In many applications – e.g., in controlling a car or a plane – reaction time is of importance. So, it is desirable to perform computations as fast as possible.

A deep neural networks has many layers which work one after another. Some of these layers perform linear combination, some apply the activation function. Thus, the time needed for the deep neural network to produce the result is much larger than for the traditional neural network. How can we save time?

- There is not much that can do to speed up the computation of a linear combination: we already apply the fastest possible algorithms for this.
- However, the time needed to compute an activation function differs: some nonlinear functions are faster to compute, for other, computations require a much longer time.

So, to save time, a reasonable idea is to select an activation function which is the fastest to compute.

Which functions are the fastest to compute: a general reminder. In modern computers, only a few operations are hardware supported: namely,

- minimum $\min(a, b)$,
- maximum $\max(a, b)$,
- sum $a + b$, and
- product $a \cdot b$.

Everything else is implemented as a combination of these elementary operations.

For example:

- When you ask a computer to compute $\exp(x)$, it actually computes the sum of the first few terms of the Taylor series of the function $\exp(x)$:

$$\exp(x) \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^N}{N!} \quad \text{(for some } N\text{)}.$$

- Division a/b is computed as $a \cdot (1/b)$, and the inverse $1/b$ is computed by an iterative procedure that consists of several additions and multiplications.

A comment about multiplication. Actually, even multiplication – while hardware supported – is actually implemented as several additions, as a result of which multiplication is the slowest of the hardware supported operations.

For example, if we multiply 7 by 11 – i.e., in binary terms, multiply 111 by 1011, we – as usual – multiply 111 by each of the digits, and then add the results:

```
   111
X 1011
------
```

```
     111
    111
+ 111
 -------
 1001101
```

As a result, we get the binary number

$$1001101 = 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 64 + 8 + 4 + 1 = 77,$$

i.e., exactly the number $7 \cdot 11 = 77$.

Which activation functions are the fastest to compute. Whatever we compute consists of the hardware supported operations.

- The more operations we perform, the longer it takes.
- So, to make computations faster, it is desirable to use as few operations as possible.

Let us therefore try by using just one such operation.

In these operations, we can use the input x and some constants c. Which constants c are the fastest to generate? Usually, 0 is the easiest value to implement: in most computers, this is default value of all numerical variables before we assign any values. For example, when you define a numerical array in Java, its values are automatically 0s. From this viewpoint:

- the constant $c = 0$ does not need any additional operation to implement, while
- all other constants require an additional step of assigning the value c to some computer cell.

So, $c = 0$ leads to the fastest computation.

In view of this, let us consider all three hardware supported operations one by one.

What if we apply minimum. If we apply minimum, we have the following options:

- we can get $\min(x, x)$ – which does not make sense, since it is simply x;
- we can get the minimum $\min(c, c')$ of the two constants – which also does not make sense, since it is simply equal to one of these constants;
- finally, we can use $\min(x, c)$.

For the simplest possible value $c = 0$, we get $\min(x, 0)$. This function is linearly equivalent to the rectified linear function, since

$$\min(x, 0) = -(\max(-x, 0)),$$

and thus, leads to the same neural network results – since before and after an application of the activation function, we have linear transformations anyway.

What if we apply maximum. If we apply maximum, we have three similar options:

- we can get $\max(x, x)$ – which does not make sense, since it is simply x;
- we can get the maximum $\max(c, c')$ of the two constants – which also does not make sense, since it is simply equal to one of these constants;
- finally, we can use $\max(x, c)$.

The simplest case is $\max(x, 0)$ – which is exactly the rectified linear activation function.

What if we apply addition. If we apply the sum, we have three similar options:

- we can get $x + x = 2x$ – which is a linear function, and we need the activation function to be non-linear – otherwise:
  - if the activation function is linear, all this neural network will compute are linear functions, while
  - many real-life processes are nonlinear;
- we can get the sum $c + c'$ of the two constants – which also does not make sense, since it is simply another contant;
- finally, we can use $x + c$ – which is also a linear function.

What if we apply multiplication. We do not consider multiplication, since, as we have mentioned, multiplication takes longer than addition, minimum, or maximum.

Conclusion of this section. It is desirable to select an activation function that can be computed as fast as possible. This leaves us with two choices:

- the rectified linear function $\max(x, 0)$, and
- a linearly equivalent function $\min(x, 0)$.

So, we have indeed provided an explanation for the use of rectified linear activation functions.

3 Selection of pooling

What is pooling. One of the main applications of neural networks is to process pictures. In a computer, a picture is represented by storing intensity values – or, for color pictures, intensity values corresponding to three basic colors – for each pixel, and there are millions of pixels. Processing all these millions of values would take a lot of time.

To save this time, we can use the fact that for most images:

- once we know what is in a given pixel,
- we can expect approximately the same information in the neighboring pixels.

Thus, to save time, instead of processing each pixel one by one, we can combine ("pool") values from several neighboring pixels into a single value.

Which pooling operations are used. At present, the most efficient pooling operations are:

- *max-pooling*, when we combine two values a and b into a single value $\max(a, b)$, and
- *averaging*, when we combine two values a and b into their arithmetic average $(a + b)/2$; this is almost equivalent to *sum-pooling*, when combine two values a and b into their sum $a + b$.

These operations are selected because they work well. As we have mentioned earlier, it is, in general, desirable to come up with a more convincing explanation for this selection.

Numerical example. Suppose that the original values are $a = 1.5$ and $b = 1.7$. Then:

- max-pooling replaces these two numbers with their maximum

$$\max(1.5, 1.7) = 1.7;$$

- averaging replaces these two numbers with their average

$$\frac{1.5 + 1.7}{2} = \frac{3.2}{2} = 1.6; \text{ and}$$

- sum-pooling replaces these two numbers with their sum $1.5 + 1.7 = 3.2$.

Our explanation. The whole objective of pooling is to speed up data processing. From this viewpoint, we need to select a pooling operation which the fastest to perform.

As we have argued earlier, this means that we need to select a pooling operation which is performed by using the smallest possible number of hardware supported computer operations – and these operations should be the fastest. The fastest is to use only one hardware supported operation, then we get $\min(a, b)$, $\max(a, b)$, and $a + b$ – exactly the two empirically successful pooling operations plus an additional operation $\min(a, b)$ which is similar to max-pooling.

Thus, we have indeed provided a reasonable explanation for the current choice of pooling operations.

4 What about fuzzy?

Let us apply the same ideas to selecting "and"- and "or"-operations. Let us apply the same analysis to "and"- and "or"-operations in fuzzy logic.

Case of "and"-operations. Among all possible "and"-operations, the fastest (= performed by a single hardware supported operation) is $\min(a, b)$ – since:

- $a + b$ does not satisfy the condition $f_{\&}(1, 1) = 1$, and
- $a \cdot b$ takes longer time than $\min(a, b)$.

The next fastest – while still performed by a single hardware supported operation – is $a \cdot b$.

Case of "or"-operations. Among all possible "or"-operations, the fastest is $\max(a, b)$ – since:

- $a + b$ does not satisfy the condition $f_{\vee}(1, 1) = 1$, and
- $a \cdot b$ does not satisfy the condition $f_{\vee}(0, 1) = 1$.

Conclusion of this section. So, we get yet another explanation of why $\min(a, b)$ and $\max(a, b)$ are empirically successful in many applications.