Machine Learning in the wild

Finally! Your new job as an intern! It is your first day at a small startup, the Pneumanauts, the only company that seemed to have some need for a young and brilliant student like you.

The boss welcomes you in person, and introduces you to the company. He tells you that measuring tires is an annoying and cumbersome routine, however a tire with wrong pressure might at best cost you just some money, as the car will consume more gas than necessary, but at worst might explode while you are driving with 120 mph on a highway. So it is a good idea to check the measurement once in a while. This is not a very big problem for a car driver, he explains, but what if you are driving a truck with dozens of wheels? It is probably nothing a truck driver is looking forward to after having spend endless hours on the roads. Would it not be nicer to have an automatic system that measures the pressure just when you drive past some device? One could deploy it at the entrance of a car park or some gas station and could warn the drivers automatically, if their tire pressures are not ok.

So the idea for a company was born, your boss threw in hard cash and some month later some ingenious engineers did build a high-tech device with an array of pressure-sensitive sensors:

![Device Diagram]

This device is placed below the surface of the street (you need to tear up the road, one of the technicians assures you). When a vehicle drives over the device, each sensor will measure the pressure at that point with a rate of 1000 Hz. As there are 100 sensors in the array, the device produces 100000 numbers per second. So the tire of a vehicle will leave some kind of footprint, that looks like this:
But the engineers were complete clueless of how to determine the pressure from such a footprint. They strongly believe that this is possible, but were not able to find some nice formula. After all, a footprint consists of roughly 100,000 numbers, way too much to really comprehend what is going on. And to make things worse, such formulas would need to account for dirt on the tires, or for rain and snow. Coming up with some kind of formula seems way too complicated.

One of the technicians read in some magazine that machine learning can handle such problems, and as they understand absolutely nothing about it, this is where you come in. They have high hopes, that you, the machine learning student, will be able to get their system working by some smart algorithms. Yes, indeed, you have attended to several courses on machine learning, and you are eager to put your knowledge into some system, so you do not mind being paid worse than an *Ein-Euro-Jobber*, so you roll up your sleeves and get to work!

You start first with analyzing the situation. For every tire of a vehicle that drives over the system, a footprint is generated. And from that you need to estimate the pressure, so it will be some function. Obviously you want the computer to learn this function. For that you need to tell the computer what to expect from a given footprint, so you show it pairs, consisting of a footprint of a tire and the pressure of that tire, i.e. a set of say \( N \) pairs

\[
D = \{(\text{Footprint}_i, \text{Measured pressure}_i)\}_{i=1,\ldots,N}.
\]

The computer should **learn** now this data, i.e. it must find a function

\[
f_D : \text{Footprint} \rightarrow \text{Estimated pressure},
\]

that will estimate (or predict) the pressure you gave him. This search is either called **training** or **learning**. You again realize that this kind of learning is nothing more than finding a good function that fits best to the given data,
– so it is actually an optimization problem than building a highly intelligent system that somehow senses and adapts to its environment.

Altogether this is called a supervised learning problem: You told the machine via several examples what you expect from a footprint, by providing it not only the footprint, but also the real pressure, which is often called a label. So in some ways you acted as a supervisor. And you want the system to give you back a real number. This is called a regression problem.

As a machine learning adept you instantly know that you want to deploy a nice neural network. For you this is a black-box, you can remember some of the mathematics behind it, but do not readily recall them. But neural networks are quite common and are one of the off-the-shelf algorithms for such problems, so it is very easy for you to find a good library.

And a few days later, you have a nice prototype that seems to be able to handle the problem! Eagerly you start to feed your system with real data the technicians have acquired meanwhile. Your very first training looks quite promising on the screen– the neural network predicts all the training examples with an incredible precision! Before you can think about the results, your coworkers impatiently snatch the software from your hands and deploy it via remote connection on the system of a customer! You feel a bit blindsided, after all you merely just hacked some kind of prototype and have no good guess how reliable that system will really work in the wild. But your coworkers just ignore your diplomatic pleas.

Already on the next day you get a visit from your boss, who looks quite disturbed– the bad feelings you had were justified. Your boss promised the customer the best working system ever, but the results of the system were rather bad! The customer claims that the performance is worse than what was promised to him, and also worse than the accuracy you had during the training.

Luckily the technicians did also gathered data from the system, and so you feed it into your network and to recheck the numbers. Just as you expected, the numbers do not look good. After looking over the data, you realize, that there are a few, very similar footprints in the training set you acquired at the company and from the customer, but the results on these differ a lot. So it seems that some noise can impact the estimation heavily.

The system probably stuck too close to the training data, you ponder, not generalizing its knowledge. This is called overfitting. You know that in its most extreme form overfitting is nothing else than just memorizing the data. This way any system is able to achieve outstanding performance on
the training set, but will fail miserably on any unseen data. Memorizing is thus not worth much, especially when the measurements are quite noisy, and footprints are so indeed. Besides, the machine would need a truly gigantic memory if it were to store all those images!

But even some overfitting is still overfitting and such a system is bound to show bad performance in the wild. How can you prevent this? You remember some kind of paradox from your lectures: the best model is not the one that predicts the pressure best on the training set! No, the best model is the one that works well on unseen data, or in other words, which can generalize best. But how can a system produce good prediction on data it never saw before?

Well, from what you have learned in your classes, basically you just simulate the situation: You set aside part of the training set, which you call validation set, and do the training only on the rest. You need some other parameter that somehow controls the complexity of the model. This is called regularization and any such parameter is generally called a hyperparameter.

The more you regularize the easier the learned model will be, thus preventing overfitting. But you must also take care not to regularize too much: A too simple model might underfit the data. This means that your model is way too simple for the complexity of your problem. After all, one valid model is to always predict the very same pressure, regardless of the actual footprint! This is the most easiest model, it is constant. Obviously no one, especially no paying customer, would want any such system. As often in life, the truth is somewhere in the middle.

So you vary your regularization parameter $\theta$ and compute the performance on the validation set, and take that $\theta$ that works best on the validation set. Bingo, your system now performs well on the training set as well on the test set. Although the results are quite good, you still have the feeling that the system should have been tested more thoroughly. No chance! Your boss forces you to deploy the system immediately at the customers site. After all, the customer has a lot of $$$ and the company cannot live just from testing systems! The technicians install the system on the very next day, and, by coincidence, there is a big inauguration ceremony for the whole system at the customers site, with all the reporters and big bosses, to show off how great the tire measurement system is. Obviously you were not invited.

The ceremony went well, and from what you gather, the customer seems to be happy with the new system. But it does not take long, and find yourself sitting in the office of your boss, this time being yelled at. The system got
installed at several other locations, and one of the new customers is very unsatisfied with the system, and yes, its your fault. Who else’s? You get 48h to make the customer happy, your boss cries, before sending you back to your office.

Good heavens, what is broken now? You start inspecting the data from the customer and you soon realize that most of the footprints look completely off— the tires are simply broken. Some of them have incredible high peaks, while others have shapes that do no resemble anything you ever saw before. Though this seems only to affects a few percent of the tires it does have an impact on the overall performance. After all the guys from marketing do promise system with 90% performance. Having 85% is way off.

But how your system could have produced good numbers on bad tires, you wonder? Gathering tires with broken footprints is not easy, they are very rare, even at the unsatisfied customers site— and even if you did find enough data to train your network, most of the tires will probably be broken in different ways. Is the measurement of a broken tire even reliable? And training with a lot of good tires and a few bad ones might still produce suboptimal results, as the system will probably learn only the good ones in order not to overfit. Or the overall performance will go down, as the system will try to adapt to the broken ones.

After thinking about this, you realize that a broken tire is a broken tire. You cannot and do not want to measure these. You need some kind of mechanism to sort these out. With a smile on your face, you notice, that the marketing people (and hopefully your boss!) will be also quite happy with this solution: After all a perfect system should be able to detect broken tires, and gives the customer a good hint to take a close look at the tire. Another reason to buy your very fine product.

Still, how to decide whether a tire is broken? This is now an outlier detection problem. Basically you want a function like this:

\[ f : \text{Footprint} \rightarrow \{\text{good, bad}\}. \]

As there are two classes, this is a binary classification problem. You need to decide to which class the footprint belongs to. If you had labels, e.g. by browsing through the footprints and deciding which of these are broken or not, this would be a supervised problem. But this is tedious and also sometimes it is not really clear to you if a footprint is good or bad. So for now you only have a bunch of unlabeled data and you roughly suspect that
there are many good and a few bad ones. Posed this way, this is now an unsupervised learning problem.

What you assume here is that there are regions in the ’space of footprint images’ in which the good ones lie and that there are regions that are occupied by bad tires. The computer should now analyze the data for you and tell you, if there are such regions.

If only the data was two-dimensional! Then you could just plot all points and see for yourself if such regions exist. But the data is high dimensional, and you, as a human, are not very good in visualizing such spaces, basically everything beyond the third dimension confuses you.

You wonder, if you can somehow project the high dimensional data into two dimensions, so that the distances between the original points are somehow preserved. That sounds awful complicated and you would need to compress the whole information of a footprint somehow into just two numbers.

Suddenly a thought flashes through your mind: Cannot the computer do this for you? You just tell it you want a faithful two dimensional view of your data. This is called dimension reduction and again is an unsupervised problem, as there are no labels involved. The task is to find a way to get the high dimensional data down into two dimensions, in a way that preserves the distance between the points as much as possible. So you start with that first, and if you are lucky, you will see already if there are two such regions. Again you find a nice algorithm, and some hours later you get your first results, that you find very interesting:

You clearly see some kind of structure! Thats awesome! And you believe that big blob in the middle are the good ones. But you still need some kind
of function that tells good ones from the bad ones. So you take one of the outlier detection algorithms, luckily the software package has several of those ready for use, and run it on the data.

After the system has learned what could be an outlier, you sort these tires out and only perform the pressure estimation on the good ones. Indeed, that works and the overall performance is much better now. Actually, you are even a little impressed by yourself.

The system has now an overall performance of over 90% and it can sort out bad tires. As soon as the technicians hear about your improvement, they pry the system out of your hands and install it on the system of the customer.

Only a few days pass by, and you get a direct call from that customers boss: If you can sort out bad tires, he asks you, why do you not tell us what is broken? Is it a broken profile, so the tire needs to changed? Or is it just some dirt? He lists several reasons, many of those you, as a layman simply do not understand. But then again, this sounds reasonable. After all, as you were visualizing the images in two dimension, you did see some kind of small clusters. Probably these clusters relate to different defects?

Again you have a nice task for you computer to solve: Now it should find a function of the form

\[ f : \text{Footprint} \rightarrow \text{Type of defect}. \]

This is now a multi-class unsupervised classification problem. It is called clustering. You remember that in practice you have to tell the computer how many clusters are there, so you have to give the computer some prior knowledge. This should be not too hard to cluster, you guess. Roughly you see five clusters, that is all the computer should need to know to find the clusters.
Luckily the machine learning software package you are using has also several routines for clustering, so you use some of the standard algorithms—the results are not that bad:

But they are not optimal either. The center class goes way down to the bottom, and instead of splitting the clusters in the top right corner, the fifth cluster is in the lower right corner, where actually nearly none of the data lies! You try again and again with different parameters and even different algorithms, but none of the packages seem to get it right. Sometimes even the big cluster is split into two halves, and the smaller ones always do seem to be suboptimal. After an whole afternoon, you give way to despair.

What to do? Well, if unsupervised clustering, at least these routines you tried, did not work well—maybe supervised clustering will help? What if you take out a few points and labeled them? Then you could run a supervised learning routine on this subset of data and still get the boundary routines? So you desperately pick a few points that seems to be representative and label them, before feeding them to a neural network. You actually doubt, that with so few data you will really get good classification results. But you still give it a try.

As you hoped for, the clusters themselves have been separated correctly. But the decision boundaries are still not lying as they should. With respect to the labeled points they look good, but you see that many of the unlabeled data would get classified into the wrong cluster. The supervised training algorithm simply did not see those.

How to deal with this problem? You could label even more examples, but then again, it is not only very annoying, you also wonder what happens, if another customer pops up? Label again hundreds of footprints?
You meditate over the problem. The idea of labeling some points is quite good— but you know you have thrown too much information away, so the supervised training cannot succeed. After all, these unlabeled points give you knowledge how to separate the clusters! You as a human see this instantly. So is it not also possible to let the computer do the very same analysis? Maybe the computer can base its classification on the labels, so the clustering itself is good, but use the many unlabeled examples to make its decision boundary better in these areas it does not have any labels?

Browsing the manual of the software package, and you find yourself quite lucky: There are some algorithms that can just do this! The problem is called semi-supervised classification, and as in the supervised case, it will find you a map of the form

\[ f_{D_l, D_u} : \text{Footprint} \rightarrow \text{Type of Error}, \]

but you provide two kinds of data: you have some, probably very few, labeled data \( D_l = \{(x_i, y_i)\}_i \) and also some, probably many more, unlabeled data \( D_u = \{x_j\}_j \).

You plug this algorithm into your system, and a few hours later you are ready to re-run your tests: The results are incredible, and it can detect not only if a tire is broken, but also determine the type of error. Actually you do not have any names for the errors yet, and even do not really know if these clusters really correspond to the types of errors the customer told you. This is up to the customer, and if he is not satisfied, maybe can one work out another clustering.

The systems works. Finally! As always, the technicians installed it at the customers site, and after a few days you receive positive feedback. Everything seems to fare quite good. The customer has baptized the errors with some fancy names, so your clustering was not off that much, and is now more than pleased with the overall performance.

Good for you! You hope that there is no more demand for changes, as your time as an intern is nearly over. Only two more days and off you go, back to your good old university. Needless to say, that there are many things one could improve on. What about weather changes? You have not really tested the system with snow and ice. Luckily, there is some kind of robustness in the system and probably it will simply detect more broken tires in winter, when there is snow on the streets. And what if it gets incredibly hot, and the tires will be so much warmer than normal? Actually you believe one should
build an adaptive system, a model that can be trained online. Ideally the system should get every now and then a labeled example and will learn on the fly by changing its internal model. So there would be no need to gather the data beforehand, but only do some measuring here and there to ensure that the system still works. Even more, why not connect all the systems and let them exchange their data? This way every measurement could be helpful for everyone and not just for the one customer.

But all these fancy ideas are up to someone else. Your time is nearly over, tomorrow it is your final day. So you start wrapping up your stuff, as suddenly your boss pops up, again. He has a bunch of rough sketches in his hand and talks something about dogs. What? You listen more closely to his blabber and figure out that his little daughter seems to nag him about getting a dog. And that he fears that his daughter is not up to the task of taking care of a dog. As a savvy entrepreneur he saw the opportunity to make a virtue out of a necessity, and wants now to build robot dogs. There is much $$$ in it, he repeats several times. And before you go, and take with you all your knowledge, he wants you to explain to his engineers how to train such a robot dog! As always, the engineers already built a prototype in some cloak-and-dagger night shift, but the robot dog is just incapable of showing any behavior at all. Again a clever software is needed.

Obviously, for your boss, training a machine learning system is the same as programming a robot, after all they both do something smart. But you know that programming a robot is a different beast altogether. There is no real label you can give the robot.

Instead, the robot can take certain actions. A robot dog, you imagine, can probably just stroll around, bark, sleep, play with a child and even pee, i.e. dropping its old oil. Now the robot dog lives in an environment. And as every robot, it has some sensors to pick up what is going round him. And it is free to act.

So how to teach the robot dog to behave nice, so that the owner is pleased with it? Clearly, there are actions that the owner might not care about. If the dog strolls around during the day, this is something normal every real dog would also do. But if it barks at night for no good reason, eventually the owner will wake up, be mad at the dog and yell at him.

Being yelled at is no good thing, so the robot dog should figure out by itself, why it was being yelled at. No sane human will try to explain to the robot dog, what it did wrong. Besides, robot dogs have no ears anyway, thanks to your boss being a niggard.
On the other hand, if there is a thief in the house, then barking at night is a good thing and the dog will get a reward, probably some new oil or just a smile. One cannot really call this feedback a label, but it is still something similar.

So the only thing the robot dog really gets back from the environment is either a reward or some punishment. But in most cases the environment is somewhat ignorant of it being there.

Unfortunately, the punishments often do not come directly. Peeing at the couch during the night will yield a punishment only after the owner detects it the next morning. This adds to the complexity of the task, as the robot has to relate past events to the current situation.

The whole setup is called reinforcement learning. In brief terms, its all about an agent living in an environment, which itself is in a state. The agent has some way to sense that state, albeit only some part of it. And the agent can take actions. These will sooner or later yield rewards or punishments— or nothing at all. The agents job is to maximize the reward. It should learn a policy so that, given a specific state of the environment, it takes those actions take will reach the goal of maximizing the reward.

You do not really believe that the engineers will be able to program such a robot dog, that is a quite a hard task. But for now that is none of your business anymore.