Machine Learning Course - CS-433

Neural Nets – Basic Structure

November 26, 2019

Last updated on: November 25, 2019

EPFL

Outline

We started this course with a basic setup. We are given a training set $S_t = \{(y_n, \mathbf{x}_n)\}$ and our aim is classification. We have seen that simple linear classification schemes like logistic regression

$$p(y \mid \mathbf{x}^\top \mathbf{w}) = \frac{e^{\mathbf{x}^\top \mathbf{w} y}}{1 + e^{\mathbf{x}^\top \mathbf{w} y}}$$

can some times work very well but they have their limits.
The key to improving such schemes is to add well chosen features to the original data vector. E.g., assume that our data is two-dimensional, where all data with label $y = 0$ lies inside the unit circle and all data with label $y = 1$ lies outside the unit circle. A linear scheme, limited to the original input, cannot classify this data well. But if we add the features $\mathbf{x}_1^2 + \mathbf{x}_2^2$ and the constant to the input then the linear classification becomes trivial.
In "real" applications we are faced with the problem that we do not know a priori what features are useful. One option is to add as many features as possible. E.g., we could add all polynomial terms up to some order to our feature vector. But this quickly becomes computationally infeasible and can also lead to overfitting.
One way to address the computational issue is to use the "kernel trick." Alternatively, we can only add very few new features but have them desiged by a domain experts.
But would it not be nice if we could *learn* the features from the data in the same way as we learn the weights of the linear classifier? This is what neural networks allow us to do.

There is currently a lot of excitement about neural networks and its many applications. At the end of this short tutorial you will unlikely be able to program a neural net to play Go like a grandmaster. Many small tricks and lots of patience and computing power are needed to train neural nets for complicated tasks. But you will be able to write small scripts to solve standard handwriting recognition challenges. We will focus on basic questions.
We highly recommend the web tutorial by Michael Nielsen, neuralnetworksanddeeplearning.com and we will follow it in many aspects.

The Basic Structure

Let us look at the structure of a neural network. It is shown in Figure 1. This is a neural net with one *input* layer of size D, L *hidden* layers of size K, and one *output* layer. It is a *feedfoward* network: the computation performed by the network starts with the input from the left and flows to the right. There is no feedback loop.
As always, we assume that our input is a D-dimensional vector. We see that there is a node drawn in Figure 1 for each of the D components of $\mathbf{x}$. We denote these nodes by $x_i^{(0)}$, where the superscript (0) specifies that this is the input layer.
The same network can be used for regression as well as classification. The only difference will be in the output layer.
Let us discuss the exact computation that is performed by this network. We already described the input layer. Let us now look at the hidden layers. Let us assume that there are

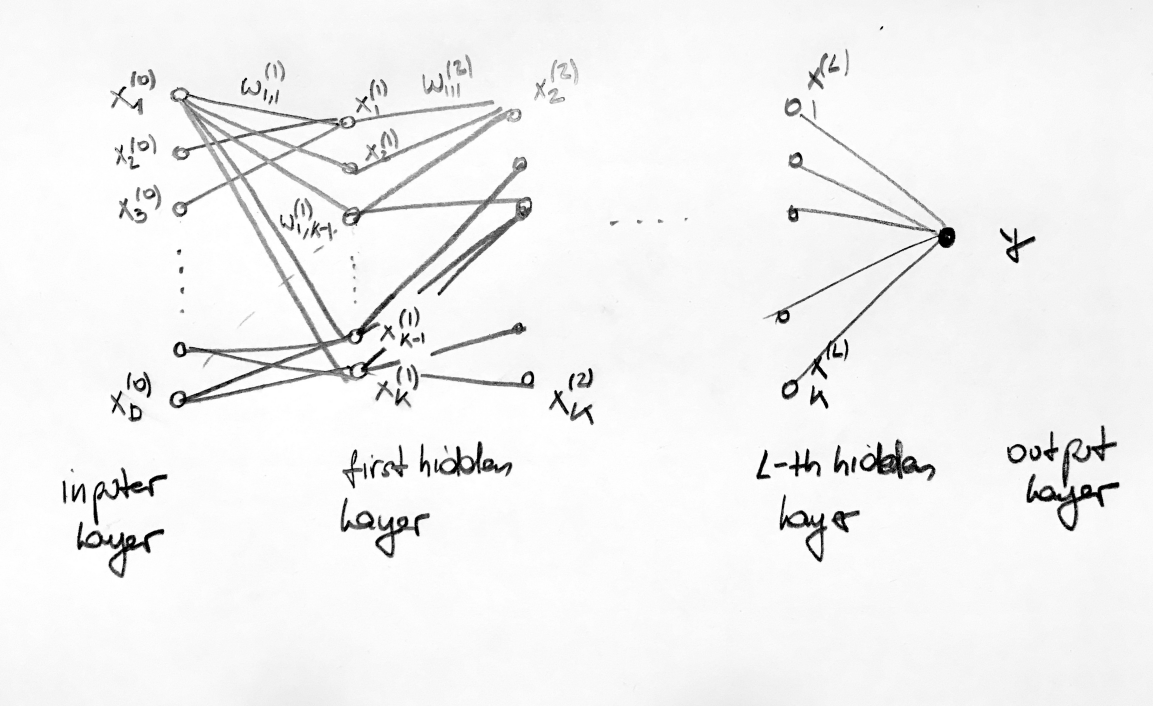

Figure 1: A neural network with one input layer, L hidden layers, and one output layer.

K nodes in each hidden layer, where K is a hyper-parameter that has to be chosen by the user and can/should be optimized via validation. There is no reason that all hidden layers should have the same size but we will stick to this simple model. How many layers are there typically? Not long ago, typical networks might have just had one or a few hidden layers. Modern applications have "deep" nets with sometimes hundreds of layers. Training such deep nets poses new and challenging problems and we will have more to say about this later.

Each node in the hidden layer l, $l = 1, \cdots, L$, is connected to all the nodes in the previous layer via a weighted edge. We denote the edge from node i in layer $l-1$ to the node j in layer l by $w_{i,j}^{(l)}$. The super-script (l) indicates that these are the weights of edges that lead to layer l.

The output at the node j in layer l is denoted by $x_j^{(l)}$ and it is given by

$$x_j^{(l)} = \phi\Big(\sum_i w_{i,j}^{(l)} x_i^{(l-1)} + b_j^{(l)}\Big).$$

In words, in order to compute the output we first compute the weighted sum of the inputs and then apply a function ϕ to this sum.

A few remarks are in order. The constant term $b_j^{(l)}$ is called the *bias term* and is a parameter like any of the weights $w_{i,j}^{(l)}$. The *learning part* will consist of choosing all these parameters appropriately for the task. The function $\phi(\cdot)$ is called the *activation* function. Many possibilities exist for choosing this function and we will explore some choices later one. For now, let us just mention one of the most popular one, namely the *sigmoid function* $\phi(x) = \frac{1}{1+e^{-x}}$. We have encountered this function already. It is the same as the logistic function. For future reference, Figure 2 shows a plot of this function. Note that the function is increasing from 0 to 1 and that for very small (negative) and very large (positive) values of the argument the function is very flat. As we will discuss, this will cause troubles when training the net since the derivative will vanish there.

It is crucial that this function is *non-linear*. Why is this? Assume not, then the whole neural-net would just be a highly factorized linear function of the input data and there would be no gain compared to standard linear regression/classification. Although it is possible to choose different activation fuctions for different nodes, it is common to choose the same function within the network.

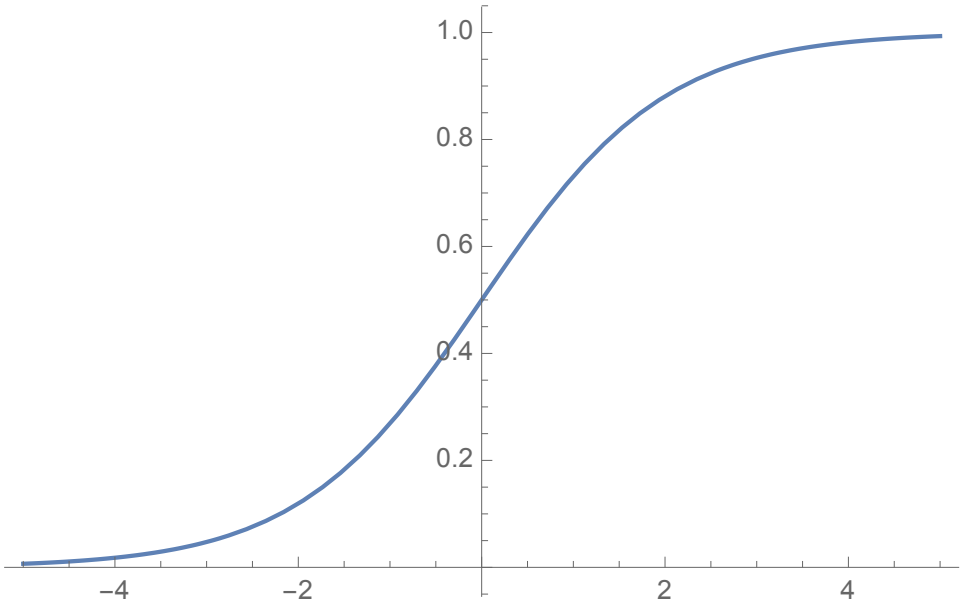

Figure 2: The sigmoid function $f(x) = \frac{1}{1+e^{-x}}$.

To connect back to our previous discussion we can decompose the neural network into two parts. The first part comprises the input layer and all the L hidden layers. The task of this part of the net is to transform the orignal input into a more suitable representation. In other words, this part of the net represents a *function* from $\mathbb{R}^D$ to $\mathbb{R}^K$. It performs the task that typically was done by domain experts, namely finding suitable features of the input. We will soon discuss what functions such a network can implement. We will see that this simple structure can approximate any continuous function arbitrarily closely provided only that we allow K to be sufficiently large.

Now that we have (hopefully) a suitable representation of the data, the final layer performs the desired ML task. This means that it is either our trusted linear regressor or perhaps a linear classifier. Presumably at this point the regression/classification task is easy. So a simple linear regres-

sor/classifier suffices.