Lecture notes including assignment 2
Supervised learning & Feed-forward networks

Alexander Mathis, Ashesh Dhawale

February 2, 2016

Every *alpinist* knows that by following gradients one will end up on the top of mountains or, in rather unfortunate cases, merely on hills (local maxima). For optimizing functions a similar principle, called **gradient ascent** (or descent depending if optimizing means maximizing or minimizing) can be used. Let's say we want to maximize the function $f : [0, 1] \to [0, 1]$ with $x \mapsto x^2$ by the following recipe:

1. pick a small $\epsilon > 0$ and some initial point x_0
2. evaluate the function derivative of f at x_i. Set $x_{i+1} = x_i + \epsilon f'(x_i)$. Go back to 2.

If ϵ is too large, or one is too close to $f'(x) = 0$ the function could also decrease during an update. For "good" choices of ϵ, one will end up close to the maximum at $x = 1$. For f this is also the global maximum, but if a function has multiple maxima, then this method will find what is called a local maximum - not necessarily the global maximum. For sufficiently regular functions this works (if one adapts ϵ simultaneously).

The same principle can be used for a high dimensional function, i.e. $f : \mathbb{R}^2 \to \mathbb{R}$, i.e. $(x, y) \mapsto x^2 + y^2$. To find a minimum one needs to update the current estimate in the direction of the gradient:

$$(x, y)_{i+1} = (x, y)_i + \epsilon \cdot \nabla f((x, y)_i). \tag{1}$$

Can this principle be used to train the perceptron? Imagine we have a supervised learning situation with data $\left(x^{(j)}, y^{(j)}\right)$, then we can define the error as

$$E = 1/2 \sum \left(y^{(j)} - o_{w,\theta}(x^{(j)})\right)^2 \tag{2}$$

Thereby, $o_{w,\theta}(x^{(j)})$ is the output of the perceptron with parameters w, θ to input $x^{(j)}$. Now we could say that we use gradient descent to minimize the errors along the parameters of the perceptron w, θ. This is not a good idea, because typically E is a highly non-regular function (in x), which is not even continuous.

1 Graded neurons and the δ rule

Therefore, we will introduce graded neurons with continuous, even differentiable, output. Such a neuron has weights $w = (w_1, \ldots w_N)$ and a output nonlinearity $\sigma(s)$, such that the output for input $x = (x_1, x_2, \ldots, x_n)$, is given by:

$$\sigma\left(\sum_i w_i x_i\right). \tag{3}$$

Common nonlinearities are the $\tanh$ or the logistic function $1/(1 + \exp(-s))$. We can write $\sum_i w_i x_i$ compactly as $w \cdot x$ Let us now apply the gradient descent method to this neuron. For a pair of data $(x^{(j)}, y^{(j)})$ the error is given by:

$$E_j = 1/2 \left(y^{(j)} - \sigma(w \cdot x^{(j)})\right)^2. \tag{4}$$

The gradient is given by:

$$\nabla E_j = \left(y^{(j)} - \sigma(w \cdot x^{(j)})\right)\ \sigma'(w \cdot x^{(j)})\, x^{(j)} \tag{5}$$

Thus, the gradient descent method suggests that starting from an initial weight vector $w^{(0)}$, one at first picks a random sample from the training data $(x^{(j)}, y^{(j)})$ and then has to update the weights according to

$$w^{(i+1)} = w^{(i)} - \epsilon\sigma'(w \cdot x^{(j)}) \underbrace{\left(y^{(j)} - \sigma(w \cdot x^{(j)})\right)}_{\delta} x^{(j)}. \tag{6}$$

Thus, in case of an error $\delta \neq 0$ the weight gets updated by the input vector $x^{(j)}$, with learning rate $\epsilon\sigma'(w \cdot x^{(j)})$. The differential $\sigma'(w \cdot x^{(j)})$ modulates the learning rate. The sign of δ makes sure the change is in the correct direction.
This rule is often called δ learning rule and is similar to the perceptron rule we saw in the last class (but applies to graded neurons). It also has an Hebbian aspect to it, as (in one term) correlates input $x^{(j)}$ to the output $y^{(j)}$. To train a neuron with the delta rule one typically sequentially iterates through the training sample to recursively update the weights.

2 Feed-forward networks

Graded neurons can be connected to produce feed-forward networks. Feed-forward means that the information is flowing unidirectionally from input to output. These networks provide powerful tools to learn complicated input-output mappings based on training data. For instance under the name *deep neural networks* they are very much en vogue and hyped.[1] The general study of such networks is beyond the scope of this course and we refer the student to the great book by Bishop.[2]

We investigate a specific, yet powerful example with 2 layers consisting of 1 hidden layer with 3 nodes and one output node. This network depicted in figure 1.

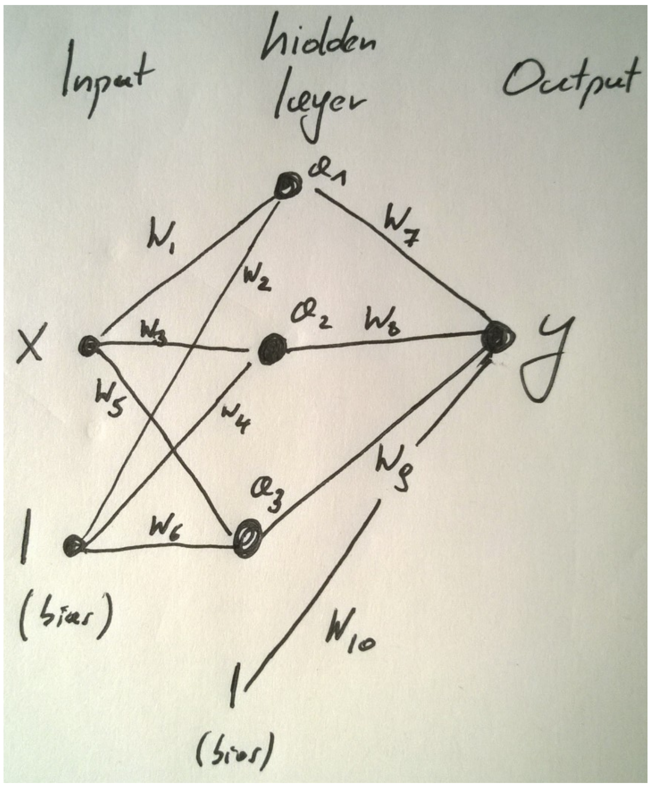

Figure 1: A 2 layer network with 1 hidden layer. The network has one 1D input x and output y. The weights w_2, w_4, w_6 and w_{10} introduce offset terms to the nodes. This is why they connect to "1".

[1]For instance the Nature News article "Computer science: The learning machines" by Nocala Jones speculates that they are bringing us close to true artificial intelligence. http://www.nature.com/news/computer-science-the-learning-machines-1.14481
[2]"Pattern Recognition and Machine Learning" by Christopher Bishop.

The network equations are given by:

$$\begin{aligned} a_1 &= w_1 x + w_2 && (7)\\ a_2 &= w_3 x + w_4 && (8)\\ a_3 &= w_5 x + w_6 && (9)\\ y &= w_7\sigma(a_1) + w_8\sigma(a_2) + w_9\sigma(a_3) + w_{10} && (10) \end{aligned}$$

We will use $\sigma(s) = \tanh(s)$. Note that $\frac{\partial\sigma(s)}{\partial s} = 1 - \sigma(s)^2$. For a data pair $(x^{(j)}, y^{(j)})$ the error is given by $E = 1/2\ \left(y(x^{(j)} - y^{(j)}\right)^2$. As usually we abbreviate $\delta = y(x^{(j)}) - y^{(j)}$. The gradient of the error is given by:

$$\begin{aligned} \frac{\partial E}{\partial w_1} &= \delta w_7(1-\tanh(a_1)^2)x && (11)\\ \frac{\partial E}{\partial w_2} &= \delta w_7(1-\tanh(a_1)^2) && (12)\\ \frac{\partial E}{\partial w_3} &= \delta w_8(1-\tanh(a_2)^2)x && (13)\\ \frac{\partial E}{\partial w_4} &= \delta w_8(1-\tanh(a_2)^2) && (14)\\ \frac{\partial E}{\partial w_5} &= \delta w_9(1-\tanh(a_3)^2)x && (15)\\ \frac{\partial E}{\partial w_6} &= \delta w_9(1-\tanh(a_3)^2) && (16)\\ \frac{\partial E}{\partial w_7} &= \delta\sigma(a_1) && (17)\\ \frac{\partial E}{\partial w_8} &= \delta\sigma(a_2) && (18)\\ \frac{\partial E}{\partial w_9} &= \delta\sigma(a_3) && (19)\\ \frac{\partial E}{\partial w_{10}} &= \delta && (20) \end{aligned}$$

Note that while the partial derivative for the weights w_7 - w_{10} are like for the single graded neuron in the last section, the other cases are different. Essentially, each partial derivative weighs δ by the weight with which it is connected to the output neuron, i.e. δw_9 for $\frac{\partial E}{\partial w_5}$. This means the errors are "backpropagated through the network". Similar rules apply for general feedforward architecture and allow efficient calculations of the derivatives, by propagating errors.[3]

With this gradient one can use the gradient descent method to train the network. From an initial weight w_0 one updates the weight vector iteratively by $w_{i+1} = w_i - \epsilon\nabla E(w_i)$ for a data point $(x^{(j)}, y^{(j)})$; the $-$ reflects that here we are minimizing the error. Again by sequentially cycling through the data one has good chances to find a good model. You will use this network to approximate a one dimensional functions $f(x) = y$ from a few samples. Image one is given a bunch of measurements $y^{(j)} = f(x^{(j)})$ for inputs $x^{(j)}$, then one can use this learning rule to approximate the data. An example of this approach for the specific function $f = \sin$ is shown in figure 2. The fit of the neural network is quite good! Its success is not arbitrary. Hornik, Stichcombe and White proved in 1989 that a network with only one hidden layer (but arbitrary many nodes) can approximate any continuous function on an interval $[-1, 1]$ arbitrarily well.[4]
Finally, a general note of caution. Obviously all limitations from the alpinist scenario apply to the backprop algorithm as well. The result might converge to a local minimum (so multiple initializations are needed), the convergence is rather slow and is not guaranteed. Despite this shortcomings in the training phase, once backpropagation gave a good network the networks can be run very fast on new input. For many problems, like face recognition, now a lot of supervised learning data is available. What seems striking about deep neural networks is that they seem to be able to digest tons of training data well, and actually "learn" to perform well.[5]

[3]See "Pattern Recognition and Machine Learning" by Christopher Bishop for more details.

[4]They actually showed that the function can be from any compact space and merely be Borel measurable. The reference is "Multilayer Feedforward Networks are Universal Approximators" in Neural Networks, Vol. 2, 1989.

[5] Check out for instance "Facebook will soon be able to ID you in any photo" by John

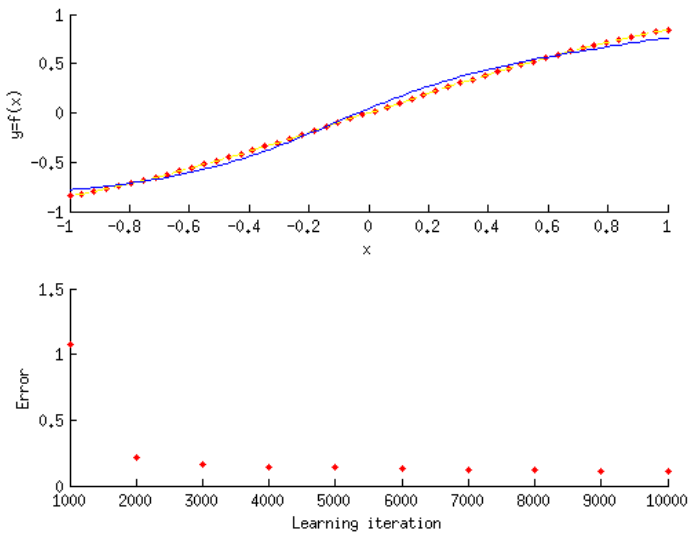

Figure 2: The upper panel shows the function $f = sin(x)$ on $[-1, 1]$ in yellow. The red dots in the upper panel are the data pairs I selected for training. The blue line depicts the input-output relationship of the network after $10,000$ iterations with learning rate $\epsilon = 210^{-3}$. The lower panel shows the evolution of the summed squared errors between the predicted and the real curve f.

3 Outlook

Unfortunately, we can not spend much more time on feed-forward networks. We hope that the few samples we served whetted your appetite for more. These networks are an exciting current topic of research and we want to recommend you a few sources if you interested in more:

- "Pattern Recognition and Machine Learning" by Christopher Bishop; Springer 2006.
- A neat online book on Deep learning http://neuralnetworksanddeeplearning.com/
- Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011. http://scikit-learn.org/

4 Assignment

1. Feed-forward network: Implement a feed-forward network with 2 layers and 3 hidden nodes. Use the $tanh(x)$ as nonlinearity.
2. Training the network: Implement the backprop learning algorithm from class, i.e. calculate the gradient ∇E_w.
3. Use this algorithm to learn the following supervised learning problems (describing the input and output pairs):
   - $\{(-1 + i/25, sin(-1 + i/25))\}_{1 \leq i \leq 50}$
   - $\{(-1 + i/25, (-1 + i/25)^2)\}_{1 \leq i \leq 50}$
   - $\{(-1 + i/25, (-1 + i/25)^{17})\}_{1 \leq i \leq 50}$
   - $\{(-1 + i/25, H(-1 + i/25))\}_{1 \leq i \leq 50}$, where $H(x) = 1$ for $x \geq 0$, and 0 otherwise.

What learning rates ϵ work? How many learning iterations do you need for good performance?

Bohannon in Science News http://news.sciencemag.org/social-sciences/2015/02/facebook-will-soon-be-able-id-you-any-photo.